Institiúid Teicneolaíochta Cheatharlach
INSTITUTE of TECHNOLOGY CARLOW
At the Heart of South Leinster

Research Report

5/12/2012

Daniel Connor

C00137906

Javascript Errors

Introduction

JavaScript[36] is a dynamic language created originally in 1995 by Brendan Eich for the netscape browser. Its primary use is in the browser to make web pages interactive, although recently it has also become popular on the server side through javascript engines such as rhino[1] and node.js. As javascript has become more popular there is a greater need for tools to help developers test their code. Projects such as phantom.js[44], which is a “headless browser”, allows the browser to be scripted using javascript. This has proven to be useful in allowing browser testing to be automated. However there aren’t many automated tools that create test cases. The ones that do are mainly oriented around security and vulnerabilities associated with javascript[6, 7] and do not focus on the quality of javascript code, or the amount of errors that occur.

Runtime errors should not occur in production software. What we want to do is to generate runtime errors in javascript to highlight places where error checking should be improved before the code is released. It should also help to highlight places where a programmer is misusing a API, whether that is a native API provided by the environment in which the javascript is running, or a third party API such as jQuery[46], Backbone[47], or underscore[48]. If we cannot cause any, it should prove that the software does enough error checking to be stable and uses any APIs correctly.

Javascript Introduction

Variables in javascript don’t have a type associated with them, similar to variables in python or ruby. They can hold any of the primitive types; string, number, boolean or undefined. They can also hold references to objects, as well as functions which are “first-class” objects[45]. Javascript uses a concept called “duck-typing” to describe the type of objects. This comes from the phrase “if it quacks like a duck and looks like a duck then it must be a duck”[27, 28] i.e. An object is defined by the properties or methods that it has, rather than a specific type. Because of duck typing, functions cannot define the types of the arguments they take. One option to get around this, is to use the `instanceof` operator, which checks whether an instance of an object is in an object’s prototype chain.

Javascript uses prototypes for inheritance rather than classes. All javascript objects are extended from the base Object, which means the prototype of Object is at the end of the prototype chain. To inherit from an existing object, the prototype of the existing object must be added to the new objects constructor[1] where the constructor is a function. An instance of a the object is created by prefixing the function with the `new` operator and invoked. When a property is assessed on an object, if the property does not exist on the object itself, the prototype chain is recursively checked for the property until the end is reached. An example of prototypal inheritance is shown below:

```
function Parent() { }
// because a function already inherits from an object
// we don't need to explicitly state that
Parent.prototype.parentFunc = function() {};

function Child() { }
// add the prototype of the parent to the child
util.inherits(Child, Parent);
// add new methods to the child
Child.prototype.childFunc = function() {};

var a = new Parent();
var b = new Child();
```

If we take a look at the structure of the resulting prototype chain, it would look something like this[2]

```
Child
    |- prototype(Child's prototype)
          |- childFunc: function() {}
          |- prototype(Parent's prototype)
                |- parentFunc: function() {}
                |- prototype(Object's prototype)
                      |- valueOf
                      |- toString
                      |- etc...
```

The example below shows an example of how `instanceof` works:

```
a instanceof Parent === true
a instanceof Child === false

b instanceof Parent === true
b instanceof Child === true
```

While in the majority of cases instanceof can be used to check what methods or properties an object has, it's not always a viable solution to use `instanceof`. Objects are not always created using prototypal inheritance and properties can get overwritten at any time. Therefore using `instanceof` doesn't guarantee that an object will have a specific property or method, or indeed

[1] The easiest method of doing this is with the node.js utility function "inherits"

[2] This is not an entirely accurate representation.

whether a property is of a specific type. This commonly leads to type errors being caused if a function does not check what properties a variable has before using them.

Javascript Errors

Errors are a common problem in the “wild” which is proven by the amount of services that have become available over recent years to keep track of errors that occur on the client side. Examples of such services are Exception Hub[41], Muscula[42] and Errorception[43]. Most of these logging services work by listening for the error event[32] that gets fired when an uncaught runtime error occurs. The error description is then logged to a remote server where the webmaster can look at the errors that have occurred while users are on their site. I asked each of these services to provide me with sample data to help analyse the most common types of errors that occur in web applications from real world use.

To get a better understanding of the types of errors that occur in javascript, we will take a short look at some of the data I obtained. To make it simpler for myself, I filtered the errors to those that came from the Google Chrome browser. Mainly because it divides up the errors into better defined types than the other browsers, so it is easier to look for the types that are most useful to us. There are a number of different types of error defined in the Ecmascript specification[17] which I will briefly explain.

TypeError: This error is thrown if an operation is tried on a variable whose value doesn’t support the operation. From our sample set of data, this was by far the most common type of error. The majority of the time errors like this occur because a function was expecting a parameter of a different type to the one it was given. A lot of the time type errors can be caused by similar reasons to reference errors. For instance, normally a jQuery plugin will add a function to the jQuery prototype. If the file containing the plugin does not load, but the plugin function is called, an error will be thrown.

```
// simplest way of defining a jQuery plugin[33]
// by adding the myPlugin function to the jQuery
// prototype. This is quite often in a separate
// file.
jQuery.prototype.myPlugin = function() {
   // this instanceof jQuery === true;
};

// create an instance of a jQuery object
var div = jQuery(“<div />”);

// if the file containing the first piece of
// code does not load this next statement will
// cause a type error because the jQuery instance
// will not have the myPlugin function.
div.myPlugin();
```

Below are some other examples of type errors:

```
var a = null;
a.a = 1;
Uncaught TypeError: Cannot set property 'a' of null

// a is not assigned to anything, so it is undefined
var a;
var b = a.c;
Uncaught TypeError: Cannot read property 'c' of undefined

var f = 1;
f();
Uncaught TypeError: number is not a function
```

ReferenceError: This error is thrown when an operation is performed on variable that has not yet been defined. This kind of error can normally be tested for using static analysis. There is one instance however, where it would be more difficult to test using static analysis and that is when the with statement is used. The with statement brings all of the properties of an object into the local scope. A reference error may occur if a property was expected on an object, but did not exist. An example is shown below. Normally this is not as common, however, because the usage of the with statement is discouraged[25], although instances of this kind of error can be seen our data. Below are some examples of reference errors:

```
var a = {
  b: 1
};

with(a) {
  // c was expected as a property of a but it does not exist
  var x = b + c;
  Uncaught ReferenceError: c is not defined
}

var b = xyz + 1;
Uncaught ReferenceError: xyz is not defined
```

RangeError: Range errors are not as common as both reference errors and type errors. They occur in Chrome[3] when a recursive function calls itself infinitely or when you attempt to create an Array with an invalid length as shown below.

```
(function f() {
   Uncaught RangeError: Maximum call stack size exceeded
   f();
})();

var a = new Array(-1);
RangeError: Invalid array length
```

Error: This error type is for generic errors that are not any of the previous types. This type of error is often extended by libraries to provide custom events. The most common environment in which javascript is run is the web browser. Web browsers provide a large number of APIs that, due to the limitations of javascript, are implemented in lower level languages. A lot of these APIs throw errors when they receive incorrect inputs.

```
document.createElement("");
Error: INVALID_CHARACTER_ERR: DOM Exception 5
```

Many more causes of errors have been introduced with “strict mode” in javascript in an effort to improve some of the problems with javascript [20].

Error Examples

Now that we have defined the different types of errors, we will give some examples of some common mistakes to bring them about.

[3] Not all browsers impose a limit on the stack size, or do but just fail silently.

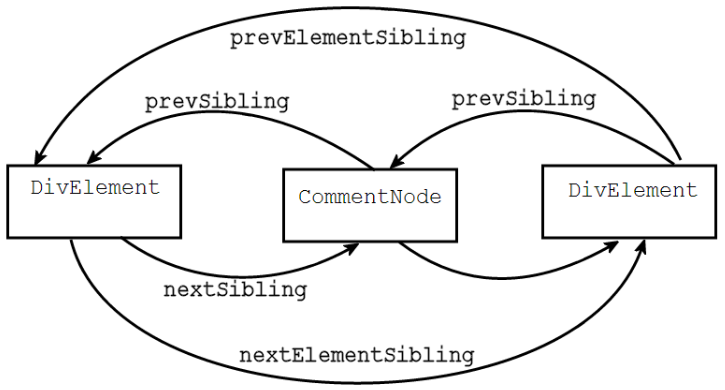
```
/**
* Gets the nth sibling node of el and if it exists adds the
* highlight class to it.
*
* @param {Node} el The DOM node to start at.
* @param {Number} n The number of nodes to traverse.
*/
function highlightNthSibling(el, n) {
  var next = el,
    i = 0;

  while(next && i++ < n) {
    next = next.nextSibling;
  }

  if(next && i == n + 1) {
    next.classList.add("highlight");
  }
}
```

The DOM(Document Object Model)[24] is a tree structure that represents a parsed html document. It provides an API for javascript to traverse it and manipulate the nodes in it. The DOM tree is made up of Nodes that provide basic functionality for traversing the DOM tree. The nodes can be subclassed to provide extra functionality, from simple divs to more the more canvas elements, both of which inherit from the `Element` class that in turn inherits from a `Node`.

Consider a snippet of HTML that looks like this:

```
<div id="first"></div><!-- This is a comment --><div></div>
```

When parsed, it will make up a set of nodes in the DOM that look like this:



In the example above, a `TypeError` may be caused due to the fact that the function `highlightNthSibling` uses the element `nextSibling` property which gives the next sibling `Node` in the DOM. This `nextSibling` property will not always reference an object that inherits from an `Element`. i.e. it could be a `TextNode` or `CommentNode`, neither of which have the same properties and functions as an element, but still inherit from a `Node`. This can easily be overcome by checking whether the returned `Node` from `nextSibling` is an `instanceof` an `Element` which is a sub-class of a `Node`. The other solution is to use the property `nextElementSibling`, which will always be a reference to an `Element`.

Here is an example of when the function will cause a type error. If we use the element with id “first” as a starting point and select the first node after it. The function will try to call the add method of the node’s classList property, which doesn’t exist on a CommentNode.

```
highlightNthSibling(document.getElementById(“first”), 1);
TypeError: Cannot call method 'add' of undefined
```

However, if we use the same element as a starting point, but set the second input parameter to 2, to select the second sibling of the element, the function will work because a DivElement has a classList property.

```
highlightNthSibling(document.getElementById(“first”), 2);
```

The resulting html string will look like this:

```
<div></div><!-- this is a comment --><div class="highlight"></div>
```

While the last example we looked at is specific to client side javascript, this next example can be caused in any javascript environment. This example might be a bit far-fetched as there will probably never be a group called “toString”, however it gets the point across. The example will cause an type error if, in the array of objects passed in, one of the objects happens to contain a group with the same name as a property on an object’s prototype.

```
/**
*  Group objects that have a group property into separate arrays
*  of objects.
*
*  @param {Array} Object An array of objects that look like this:
*   {
*      group: “Cars”,
*      name: “Ford Focus”
*   }
*  @param {Object} An object with a collection of keys made up of
*   all the groups from the array of objects.
*/
function groupObjects(objects) {
  var groups = {};

  for(var i = 0; i < objects.length; i++) {
     var object = objects[i],
      groupName = object.group,
      group = group[groupName];

    if(!group) {
      group = group[groupName] = [];
    }
    group.push(thing);
  }
  return group;
}

// this will cause an error because it doesn’t overwrite the
// toString property with an array, then tries to call the push
// method on it.
groupObjects([
  {
     group: “toString”,
     name: “something”
  }
]);
// this will work fine because none of the groups have
// the same name as anything on an object’s prototype
groupObjects([
  {
     group: “cars”,
     name: “ford focus”
  }
]);
```

For our final example we will show the problems caused by lack of types in javascript. Consider the following function:

```
function example(obj) {
  obj.func();
}
```

The function is a trivial function that merely takes an object and executes a function property of it. From the explanation of javascript types earlier, we can see that that it would be quite easy to cause a number of errors here. Here are some examples.

```
example(1);
TypeError: Object 1 has no method 'func'

example();
TypeError: Cannot call method 'func' of null

example({
  func: 1
});
TypeError: Property 'func' of object #<Object> is not a function
```

These errors are cause because of non-existent checking of the type of the parameter obj. Next we will demonstrate some examples of where there is error checking but it is inadequate.

```
function example(obj) {
  if(obj) {
     obj.func();
  }
}
```

Here we check if obj was actually passed as a parameter, which will prevent an error from calling with no parameters, or falsy parameters. e.g. `example()`, `example(null)`, `example(0)`.

It’s clear that the error checking we have is not enough. Next we can improve it by checking whether the obj parameter is an object:

```
function example(obj) {
   if(typeof obj === “object”) {
      obj.func();
   }
}
```

Again, even though our type checking is better, it is not enough. We can still cause an error by passing null as a parameter, because using the typeof operator on null gives “object”. It will prevent an error resulting from passing numbers, or no parameters. But we will get an error by passing an object that has a func property, but is not a function.

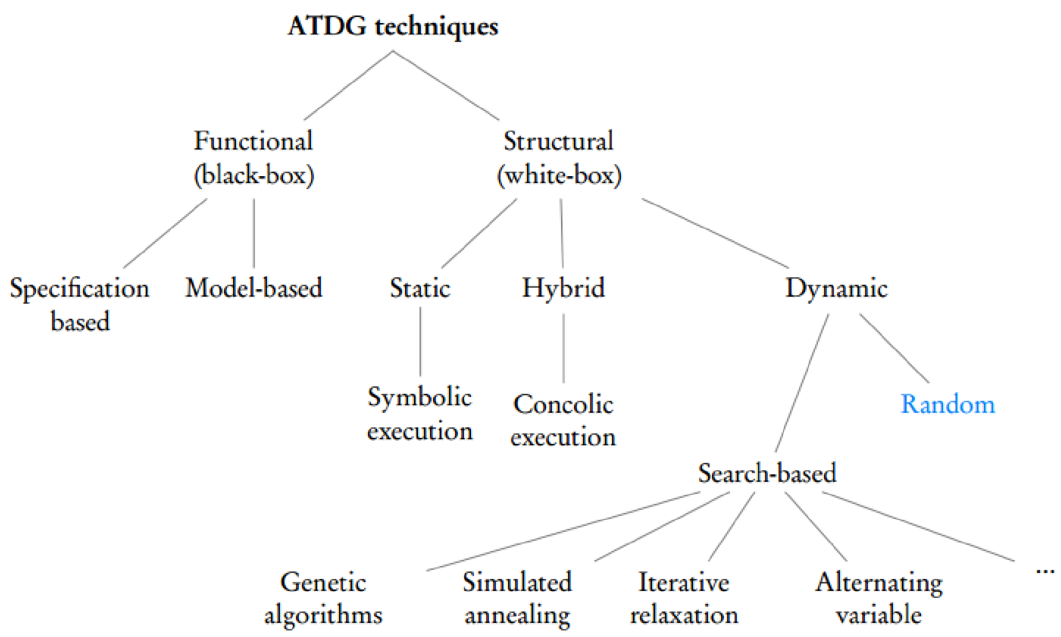
```
function example(obj) {
  if(typeof obj === “object” && typeof object.func === “function”) {
    obj.func();
  }
}
```

In this final iteration of our example function, it is impossible to cause an error, no matter what parameters we pass into the function. Although this kind of type checking everywhere may be considered too verbose, as we have stated before, it may be useful to show the programmer where he is using an API wrongly, as in the first example, or edge cases where a property may accidentally be overwritten in our second example.

Test Generation

Now that we have established the need for a Javascript error generator, we will take a look at the ways we might do this. To cause exceptions in the code, we must generate test data that can be used on functions to cause them to throw errors. There are a number of different Automatic Test

Data Generation(ATDG) methods which are outlined in the diagram below[2].



To the left of the diagram are Functional or black-box techniques. Functional testing[14] is not concerned with the internal workings or structure of the code, rather it is concerned with proving that a program provides the correct functionality.

To the right of the diagram are Structural or white-box techniques[14]. In contrast to functional testing, structural testing is concerned with the inner workings of the program. It “tries to ensure that it does not crash under any circumstances, regardless of how it is called”[2]. This is the kind of testing we are interested in, because errors are concerned with the inner workings of the code.

Test data generation methods can also be further divided into three different groups, random, path-oriented, and goal-oriented test data generation[52]. To cause errors, we need to be able to find places where an operation is executed that could cause an error. This can involve executing the code and directing it to follow as many branches as possible to find places where errors can occur. Then trying to execute that path with inputs that will cause an error. If the path cannot be executed with inputs that will cause an error, then we can be sure that an error cannot be caused at that point. This is in the goal-oriented category of test data-generation, as we need to execute a specific path in order to read a goal.

Another method is for the programmer to insert annotations into the code which define where an error can occur, then we need to try and execute that code with inputs that can cause an error. This cuts out the need for finding branches ourselves, but puts more work on the programmer. It also may be more difficult to find the path from the point of the annotation to where inputs are defined, compared to the previous method where we know the path we have taken to get to that point. Seeing as the type of errors we are looking for are type errors, to avoid having the programmer having to annotate specific areas to test, we could scan the program for places where type errors can occur.

To be able to guide the program through different paths of the code, we need to be able to generate inputs that will satisfy the constraints required to enter each branch of the path. There are three main ways of doing this that are shown on the diagram above.

Static Data Generation

Static test data generation involves the static analysis of a program to find flow of control structures that determine the path that the program takes. Each flow of control structure has a set of constraints that define whether or not the branch will be taken, or in the case of loop statements, how many times, if at all, the branch will be executed. By solving each of the constraints that “guard” a branch we can determine a set of inputs that cause the program to execute a particular path. To generate the test data, a technique called symbolic execution is used. “Symbolic execution gathers constraints along a simulated execution of a program path, where symbolic variables are used instead of actual values, such that the final path predicate can be rewritten in terms of the input variables. Solving the resulting system of constraints yields the data necessary for the traversal of that path [20, 21]”[2]. There are a number of problems with symbolic execution however[2, 49, 50].

- One of the main problems required to be solved for symbolic execution is constraint solving which is proven to be intractable [11, 12]
- the presence of input variable dependent loops can lead to infinite execution trees 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
- even if the path constraint is linear, solving it can lead to very high complexity
- it is not very good at handling dynamic types and constructs [11, 40] like those in javascript

Solutions have been proposed for some of these issues, but they can typically only handle a subset of the language and are not very useful for practical use. These solutions also increase the complexity of the implementation, which is too much of a cost for reduced usefulness. A more practical approach is using a hybrid approach which we will explain later.

Dynamic Data Generation

Whereas static methods of test data generation are purely based on static analysis of a program and the program does not executed, dynamic methods are based on execution of the program.

Random

Random test data generation is the simplest of all the methods of test-data generation and can be used to test any type of input[52]. Random testing works exactly as its name describes, e.g. for a string a random stream can be used as input. Random test data generation could be used to find errors in a program, but it has a very low probability of finding semantically small faults [53], and “although it can reach deep states of a program, it fails to be wide, that is, to capture a large variety of program behaviours”[10]. Also, as it is random, it cannot allow us to guide execution through the program to achieve our goal, which would be the optimal solution.

Search-Based
Whereas symbolic execution solves constraints using a constraint solver, **“**search-based test data generation uses heuristics to guide the generation of input data, so that the inputs are more likely to execute paths that contribute to the overall test coverage objective. This involves modelling the test coverage objective as a heuristic function or objective function, that evaluates the fitness of a chosen set of inputs with respect to a coverage objective”[2] or in our case with respect to an error that we want to cause. For example we can figure out what kind of value a variable should have based on the properties that are accessed from it. There are many different search-based techniques that could be used such as; simulated annealing [54, 55], iterative relaxation [56] and genetic algorithms [57].

Search-based test-data generation seems to be the best solution to our problem, because it allows us to guide the execution of the program and allows us to tune the inputs based on the error we are trying to cause.

Hybrid Data Generation
Concolic execution is a hybrid approach to test data generation by combining symbolic and concrete execution. In concolic execution, the program is first run using real inputs to find symbolic constraints. The constraints collected during the first run are then used to direct the execution of consecutive executions until all feasible paths are found. There are a number of examples of symbolic execution engines that exist for javascript, including Kudzu [6] and Rozzle [7]. Both of these engines are used to analyse javascript programs for vulnerabilities.

Kudzu uses a custom constraint solver that “supports the specification of boolean, machine integer (bit-vector), and string constraints, including regular expressions, over multiple variable-length string inputs” [6]. It also cross compiles the javascript to an intermediate language called JASIL to run on a modified javascript interpreter, which is a form of instrumenting the code.

There are many other examples of tools that use concolic execution because of its usefulness in real-world programs such as DART [8], CUTE [9], PEX [41], EXE [7].

As this method of test-data generation has been proven to be practical for real-world use, especially for javascript, and it allows to guide the execution of the program to achieve a goal, it seems to be a feasible solution to our problem, however, it does not give us the ability to tune our inputs like search-based methods do.

Tracking what the code is doing

From the implementations I have explained above, they require some method of keeping track of what the code is doing as it's running.

Code instrumentation

Code instrumentation involves the modification of the code of a program so that it executes it as normal, but allows monitoring of what is happening in the code at any point in time. Some applications of code coverage include Code Coverage Testing [2], Profiling[3]. Code instrumentation is useful to us in this case because it allows us to obtain a profile of the code and the paths that it takes so that we can analyze it. There are some open source javascript parsers available such as esprima[21] and uglifyjs[22]. Both of these tools allow easy extension and the ability to modify the resulting Abstract Syntax Tree(AST) and re-output the code based on the new AST.
One of the disadvantages of using code instrumentation is that language features of javascript, such as variable hoisting and scope[23], must be handled by us rather than a javascript engine. Every variable in javascript exists from the beginning of the scope in which it is defined. Its value will remain undefined, until it is given a value. An example is shown below:

```
function example1() {
  // a gets hoisted to here
  // var a;

  if(a == undefined) {
    // this code will be executed,
    // because the variable was hoisted
  }
  var a = 1;
}

// example2 can be called here even though
// it is defined later in the file.
example2();

// functions defined like this get hoisted
function example2() {};

// example3 is undefined here, so it cannot be invoked
example3();
// functions defined like this do not
var example3 = function() {};
```

Proxy API[16, 34]

Flycatcher uses a specification defined in Ecmascript 5(ES5)[17] called a Proxy. “Proxies are objects for which the programmer has to define the semantics in JavaScript”[35] rather than a lower level language like c++ as it normally would in a javascript engine. Among the use cases for proxies as defined by the specification is transparent logging, tracing, and profiling[16] which is exactly what we require. Proxies work by defining a set of functions which are invoked when an operation is performed. This allows us to track what is happening to the object, and also allows us to define what value gets returned from the operation. Proxies can only define semantics for objects and functions, but seeing as the errors we are looking for are to do with objects being of the wrong type(having different properties and functions than expected), we can use proxies instead of arguments to functions and track what properties are being accessed.

```
function example(obj) {
  // we cannot detect when an object is accessed
  if(obj) {
    // we can detect when a property of the object is accessed
    if(obj.prop) {
      // and also when a property is invoked
      obj.prop();
    }
    // we can detect when an objects properties are enumerated
    for(var i in obj) {
      // do something
    }
    // by overriding the valueof function in the proxy we can return
    // primitives. Note however we cannot use the strict equal operator
    // (===)
    if(obj > 1 || obj == 1) {
    }

  }
}

var proxy = Proxy.create({...});

example(proxy);
```

As you can see from the example above, we can track what operations are performed on an object by passing a proxy as an argument, instead of the object. However, if an object is created within the function, we cannot track what is happening to it. You can also see from the example that we can specify what value is returned from the valueof function when it is used. Its usefulness is limited to the operators that it can be used for.
While this method of tracking the code allows us to track what is happening to specific object and the operations performed on it, it doesn’t allow us to track the branches that have been executed, or evaluate the conditions required for the code to take a particular path.

Code instrumentation, while it does have some disadvantages, gives us the ability to implement all of the functionality that the proxy api does and more. The disadvantages it does have are quite easily overcome.

Bibliography

1. **B. Beizer.** *Software Testing Techniques.* Van Nostrand Reinhold, 2nd edition, 1990.

2. **Lafargue, Jerome de.** *Flycatcher: Automatic unit test generation for JavaScript,* http://www.doc.ic.ac.uk/teaching/distinguished-projects/2012/j.delafargue.pdf

3. **Mustafa M. Tikir and Jeffrey K. Hollingsworth.** *Efficient instrumentation for code coverage testing***.** In Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA), pages 86-96, 2002.

4. **Omri Traub, Stuart Schechter, and Michael Smith.** *Ephemeral instrumentation for lightweight program profiling.* In Technical report, Harvard University, 2000

5. **King, James C.** *A new approach to program testing.* In proceedings of the international conference on Reliable software archive. Pages 228 - 233, 1975

6. **Saxena, Prateek.** *A Symbolic Execution Framework for JavaScript.* In SP '10 Proceedings of the 2010 IEEE Symposium on Security and Privacy. Pages 513-528, 2010

7. **C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert.** *Rozzle: De-cloaking internet malware***.** In IEEE Symposium on Security and Privacy, May 2012

8. **Godefroid, Patrice.** *DART: Directed Automated Random Testing.* In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. Pages 213-223, 2005

9. **Sen, Koushik.** *CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools.* In CAV, volume 4144 of Lecture Notes in Computer Science, 419–423, 2006

10. **Majumdar, Rupak.** *Hybrid Concolic Testing.* In ICSE '07 Proceedings of the 29th international conference on Software Engineering. Pages 416-426*,* 2010.

11. **Tracey, Nigel.** *Automated Program Flaw Finding using Simulated Annealing.* In ISSTA '98 Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis. Pages 73 - 81, 1998.

12. **Anand, Saswat.** *Techniques to facilitate symbolic execution of real world programs.* PhD dissertation, Georgia Institute of Technology, 2012.

13. **Wassermann, Gary.** *Dynamic Test Input Generation for Web Applications.* 2008.

14. **McMinn, Phil.** *Search-based Software Test Data Generation: A Survey. Software Testing.* In Verification & Reliability archive, Volume 14 Issue 2, June 2004 Pages 105 - 156, 2004.

15. **Alshraideh, Mohammad and Bottaci, Leonardo.** *Search-based software test data generation for string data using program-specific search operators.* In Software Testing,

Verification & Reliability - UKTest 2005: The Third U.K. Workshop on Software Testing. Research archive, Volume 16 Issue 3, September 2006. Pages 175 - 203, 2006

15. **F.S. Ocariza Jr, K. Pattabiraman, and B. Zorn.** *JavaScript Errors in the Wild: An Empirical Study,* in Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on, pages 100-109. IEEE, 2011

16. http://wiki.ecmascript.org/doku.php?id=harmony:proxies [Accessed November 27th 2012]
17. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf [Accessed November 27th 2012]
18. https://developers.google.com/chrome-developer-tools/docs/remote-debugging#protocol [Accessed November 27th 2012]
19. https://wiki.mozilla.org/Debugger [Accessed November 27th 2012]
20. https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope/Strict_mode.
21. http://esprima.org/ [Accessed November 27th 2012.
22. https://github.com/mishoo/UglifyJS [Accessed November 27th 2012]
23. https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/var [Accessed November 27th 2012]
24. https://developer.mozilla.org/en-US/docs/DOM [Accessed November 27th 2012]
25. http://www.yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/ [Accessed November 27th 2012]
26. http://js-symbolic-executor.googlecode.com/ [Accessed November 27th 2012]
27. http://rubylearning.com/satishtalim/duck_typing.html [Accessed November 27th 2012]
28. http://bigdingus.com/2007/12/08/just-what-is-this-javascript-object-you-handed-me/ [Accessed November 27th 2012]
29. http://bolinfest.com/javascript/inheritance.php [Accessed November 27th 2012]
30. http://www.crockford.com/javascript/inheritance.html [Accessed November 27th 2012]
31. http://ejohn.org/blog/simple-javascript-inheritance/ [Accessed November 27th 2012]
32. https://developer.mozilla.org/en/docs/DOM/window.onerror [Accessed November 27th 2012]
33. http://docs.jquery.com/Plugins/Authoring [Accessed November 27th 2012]
34 **Tom Van Cutsem.** *Proxies: Design Principles for Robust Object-oriented Intercession APIs.* In Conference Paper. 2010
35. https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Proxy [Accessed November 27th 2012]
36. https://developer.mozilla.org/en-US/docs/JavaScript [Accessed 28th November 2012]
37. http://nodejs.org/ [Accessed 28th November 2012]
38. http://www.jshint.com/ [Accessed 28th November 2012]
39. http://www.jslint.com/ [Accessed 28th November 2012]
40. http://www.jscheck.org/ [Accessed 28th November 2012]
41. http://www.exceptionhub.com/ [Accessed 28th November 2012]
42. http://www.muscula.com/ [Accessed 28th November 2012]

43. http://errorception.com/ [Accessed 28th November 2012]
44. http://phantomjs.org/ [Accessed 28th November 2012]
45. https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope [Accessed 28th November 2012]
46. http://jquery.com/ [Accessed 29th November 2012]
47. http://documentcloud.github.com/backbone/ [Accessed 29th November 2012]
48. http://documentcloud.github.com/underscore/ [Accessed 29th November 2012]
49. **B. Korel**. *Automated software test data generation*. IEEE Transactions on Software Engineering, 1990.
50. **Meudec, Chris**. *Atgen: automatic test data generation using constraint logic programming and symbolic execution*. In Software Testing, Verification and Reliability 11,pages 81–96. 2001
51. **R. Ferguson, B. Korel**. *The chaining approach for software test data generation*. IEEE Transactions on Software Engineering, January 1996.
52. **Edvardsson, Jon**. *A survey on Automatic Test Data Generation*. In Proceedings of the Second Conference on Computer Science and Engineering, pages 21-28. October 1999.
53. **J. Outt, J. Hayes**. *A semantic model of program faults*. In International Symposium on Software Testing and Analysis, pages 195-200. 1996.
54. **N. Tracey, J. Clark, K. Mander**. *The way forward for unifying dynamic testcase generation: The optimisation-based approach*. In International Workshop on Dependable Computing and Its Applications, pages 169–180. 1998
55. **N. Tracey, J. Clark, K. Mander**, J. McDermid. *An automated framework for structural test-data generation*. In Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on. pages 285–288. 1998
56. **N. Gupta, A. Mathur, M. Soffa**. *Automated test data generation using an iterative relaxation method*. In ACM SIGSOFT Software Engineering Notes, vol. 23, ACM, pages 231-244. 1998.
57. **C. Michael, G. McGraw**. *Automated software test data generation for complex programs*. In Automated Software Engineering. Proceedings. 13th IEEE International Conference on. pages 136-146. 1998.