Institiúid Teicneolaíochta Cheatharlach

INSTITUTE of TECHNOLOGY CARLOW

At the Heart of South Leinster

Project Plan

14/12/2012

Daniel Connor

C00137906

Table of Contents

Introduction

JavaScript[1] 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[2] and node.js[3]. As javascript has become more popular there is a greater need for tools to help developers test their code. Projects such as phantom.js[4], which is a “headless browser”, allows the browser to be scripted using javascript. There are also static analysis lint tools[5, 6] that analyse the code for syntax errors or reference errors. 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[7, 8] 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[9], Backbone[10], or underscore[11]. If we cannot cause any, it should prove that the software does enough error checking to be stable and uses any APIs correctly.

The DOM(Document Object Model) API is the api that is provided by browsers to allow javascript to manipulate a web page. The DOM is made up of a tree of Nodes, which are extended to provide different kinds of functionality to a web page. Because javascript the main use of javascript is in the browser, and it’s primary use there is to manipulate web pages to allow them to be interactive, that is where we are going to concentrate our efforts.

What we aim to achieve

What we want to do is create a tool that does the following:

- find where runtime errors can occur in javascript code.
    - The majority of runtime errors are type errors. This does not necessarily mean that we need to know the path required to take to reach a certain point, rather we can statically execute the program using symbolic execution to see whether a piece of code can be reached with values that can cause an error. Basically we need to track each variable's value at each point in the code. If we are able to do this, then we can track what type of values it is possible for a function to return. More on this is explained in the Implementation section. At first we will only use a subset of javascript to prove that this method of finding errors works, then

expanding it to include a larger subset.
    - For other custom errors thrown by DOM operations, we need to define what errors can be thrown by each operation. However, due to the limited time we have, we will only define a subset to prove that our project will work.
- generate inputs that demonstrate errors.
    - We should be able to generate inputs, or a range of inputs that when used with a function cause the error identified by the first step. While at first we will just identify where errors can occur, we can use either symbolic execution or a search-based algorithm to prove that we can reach the identified point in the code with the required values.
- suggest fixes or improvements based on what we find from steps 1 and 2
    - How to fix an error that is discovered by our tool may not be immediately obvious to a programmer, from the information that we used to find the error in the first place, it should be possible to allow

Examples

The following are a set of examples functions that cause errors that we are aiming to find with our implementation. I will also describe what errors we should be able to find as well as how they should be able to find them, and generate an input that throws the error.

1.

A lot of the time in javascript it is useful to be able to set the content of an element using a string of HTML which will then be parsed by the browser and the resulting nodes will be set as the content of the element. When something needs to be done to an element that was in the string of html, a reference to the element is required. If the element is expected to exist in the HTML string but does not, an error will occur when a property of the resulting value is attempted to be accessed. In this case we present a function that creates a Form element and sets some content. It also allows a function to be passed in to act as a callback for when the last child of the form is clicked. The last child should be an button.

```
/*
* Create a form with the specified content and add an event listener to
* the last child which should be a button.
*
* @param {String} content A string of html to set as the content of the page.
* @param {Function} cb A callback function for when the last child is clicked.
*/
function createForm(content, cb) {
  // Create an element to add the new content to.
  var form = document.createElement(“form”);
  // set the content of element to be the DOM structure that is represented
  // by the string contained in content.
  form.innerHTML = content;

  // add an event to the last child node of form
  // An error may be caused here if el does not have any children.
  form.lastChild.addEventListener("click", cb, false);
  TypeError: Cannot call method 'addEventListener' of null
  return form;
}

// set the content of the body element to be a text input and a button.
createForm("<input type='text' name="name"><button>Submit</button>", function(e) {
  // do something when the button is clicked
});

// The error will be caused on this run because el will not have any children.
createForm("");
```

As you can see, the error is thrown when there is no last child on the form element. Because we need to be able to create a test input as a html string, we can generate a html string from our knowledge of what each of the Node properties represent. In this example because we need a null lastChild value to cause the error, an empty string obviously gives that result.

2.

When trying to manipulate an element, a reference to the element is required. This involves using one of the Element selector functions i.e. getElementById, getElementsByClassName, etc... If the element does not exist in the DOM, a null value is returned. Without checking for this null value, if a property of the element is attempted to be accessed, an error will be caused.

```
/**
*  A function that checks whether the element with an id is checked or not.
*
*  @param {String} id The id of the element to check.
*/
function isChecked(id) {
  // get a reference to the element by id
  var el = document.getElementById(id);

  // get the checked attribute of the element
  // if an element with id "id" does not exist
  // an error will happen here because el will be null
  return el.getAttribute("checked");
  Uncaught TypeError: Cannot call method 'getAttribute' of null
}

isChecked("doesnt-exist");
```

In this example the error is caused by trying to call a method on a null object because the result of getElementById wasn't checked for the null value. We need to rely on the information we have about the DOM API functions return values to see that the getElementById can return either null or an Element.

3.

When using a DOM selector such as getElementsByClassName, which returns a list of elements, a structure called a NodeList is returned. A node list is "live" which means it is a list that represents the values that are currently in the DOM. So when, for example, a node in that list gets removed from the DOM in another operation, the NodeList updates to represent that change.

```
<ul>
  <li class="item">
    <ul>
      <li class="item"></li>
      <li class="item"></li>
    </ul>
  </li>
  <li class="item"></li>
  <li class="item"></li>
  <li class="item"></li>
  <li class="item"></li>
  <li class="item"></li>
</ul>
```

```
/**
* Clear the contents of all the elements with the specified className
*
* @param {String} className The class name of the elements to clear.
*/
function clearContents(className) {
  var items = document.getElementsByClassName(className),
    // items.length gets evaluated once, so even if “items”
    // updates, numItems will stay the same.
    numItems = items.length;

  for(var i = 0; i < numItems; i++) {
    var item = items[i];

    // clear the element by setting its content to an empty string
    // you can see above that the first item contains a sub list of items
    // that will now be removed. “items” will update to represent this
    // and its length will be shortened by two. Because “numItems” is larger
    // than items.length, the loop will try to access elements that do not exist
    // in items anymore, causing an error here because item will be undefined.
    item.innerHTML = "";
    Uncaught TypeError: Cannot set property 'innerHTML' of undefined
  }

clearContents(“item”);
```

This example is more complicated than the other ones as it involves a loop and a NodeList which is a dynamic list that updates to represent the current nodes in the DOM. It may be possible to detect an error if we define that modifying the DOM can update the content of a NodeList.

4.

A node must exist as a child of another node to be able to remove it.

```
/**
* Remove an element from the body of the page
*
* @param {Node} child The node to remove.
*/
function removeChild(child) {
  // child does not exist as a child of the body element
  // so an error is thrown.
  document.body.removeChild(child);
  Error: NOT_FOUND_ERR: DOM Exception 8
}

// create an element that has not been inserted into the DOM
var a = document.createElement(“div”);
```

```
// try to remove the child from the body
removeChild(a);
```

In this example we can see a custom error thrown by the DOM API itself. This error occurs when the element that is an argument cannot be found in the correct location in the DOM. In this example the error will happen if child is not actually a child of the body element. In this case an input that will cause an error is anything that is not a child of the body element, so we have to depend on the information we have about the removeChild function to generate the error.

Implementation

For our implementation we are going to concentrate on DOM based errors in browsers and client-side javascript as this is the predominant use of javascript today.

Error Conditions

There are two main conditions that we need to find that can cause an error:

- A condition that is required by an API or function before a custom error is thrown.
- A condition that allows an incorrect operation to occur on a variable. i.e. A TypeError

Because, as I have stated above, we don't have to access to the DOM implementation, we need to know what errors can be thrown and what values the properties can have. One method of achieving this would be to create a DOM implementation in javascript that mimics the actual DOM. Another, more achievable, method would be to define properties of each property and function of the built in DOM objects, such as by specifying the types a property can contain and whether they can be set or are read-only. For functions, we can define what errors they throw and what kind of values they return, if any. This would be better suited for our use case because, seeing as we are only going to handle a subset of javascript, we won't be able to handle the amount of javascript required to implement the DOM. The DOM objects include Nodes and Elements and their subclasses.

There fixed number of occasions in native javascript that TypeErrors can occur, so we can define the conditions required to make them occur.

To figure out whether errors can occur we need to statically analyse the code, then evaluate it to figure out what operations happen to an object. When we get to an operation that could cause an error, we can test to see if a variable has the value necessary to cause the error. As we go deeper into conditions/if statements, we can build a "picture" of what the object needs to look like to allow the execution of the code to get to a certain point. To be able to analyse the code, we need to convert it into a program readable format, such as the one produced by the espirma[12] javascript parser. Below is a sample of how we might achieve the above:

Here we define some of the properties and functions of a Node and an Element which inherits from a Node. As I have said above, the DOM API can throw custom errors based on incorrect parameters, and as we don't have access the implementation, we need to define when these errors can be thrown. While we will need to define those in our implementation, for now here are the types of values each property can have.

```
Node: {
  nodeType: typeof === "number",
  parentNode: null || instanceof Node,
  firstChild: null || instanceof Node
  lastChild: null || instanceof Node,
  prevSibling: null || instanceof Node,
  nextSibling: null || instanceof Node,
  appendChild: typeof === "function",
  addEventListener: typeof === "function",
  removeEventListener: typeof === "function",
  normalize: typeof === "function",
  cloneNode: typeof === "function"
}
Element: {
  attributes: instanceof NamedNodeMap,
  childNodes: instanceof NodeList,
  children: instanceof HTMLCollection,
  className: typeof === "string",
  nextElementSibling: instanceof Element || null,
  prevElementSibling: instanceof Element || null,
  // The prototype of an Element is a Node which means
  // it inherits all of a Node's properties
  prototype: Node
}
```

For the following examples, after each condition, we define what value each of the variables must have to be for after the condition has evaluated to true. We also show after each condition what the input should look like to reach that point in the code.

This first example doesn't do anything useful, it just merely demonstrates how our method should work.

```
// we are required to pass in an element to this
// function, but there is nothing stopping us
// from passing in anything.
function doSomethingToElement(el) {
  // if el is a falsy value we create an element
  // as el's new value
  if(!el) {
    el = document.createElement("div");
  }

  // at this point in the code we know that el must exist
  // and cannot be a falsy value (false, null, undefined, 0, "", NaN)
  // note that any value other than null and undefined can have properties
  // i.e. they can use the dot notation to access a property and even try
  // to set it, even though it won't happen.
  // el = truthy || (!el(original) && el instanceof Element)
  // It should only be necessary to represent the largest set of values that a variable
  // can currently be. e.g. an instance of Element is within the set of values included
  // in truthy. This is possible because when we come across a piece of code, if the
  // largest set of possible values does not guarantee that an error cannot happen, then
  // we can assume that an error is possible.

  if(el.nextSibling) {
    // Once we get in here we can assume that el.nextSibling is a truthy value.
    // This means that el currently looks like { nextSibling: %truthy% } or it's an
instance
    // of an Element, which, if it was just created will have the property nextSibling
    // set as null because it cannot have another sibling as it has no parent.
    // Another option is that nextSibling can exist on el's prototype which would mean
    // that el could even be a number if Number.prototype.nextSibling was set as a truthy
    // value. This is one of the problems with extending native types.
    var next = el.nextSibling;
    // Because we know that el can be an Element that was created above
    // if the original argument was falsey and we know that nextSibling is
    // either null or an instance of a Node. We also know that el.nextSibling is truthy,
    // so next is now truthy or a Node, which is included in Node.
    // next == truthy

    // next now becomes the same as el.nextSibling which is any truthy value
    // Or else if the first if statement evaluated as true, then el.nextSibling must
    // be an Element, because it is just after being created.
    if(next.parentNode) {
      // as we nest deeper into if statements that check for properties on an object
      // we can build up a profile of what sort of object we have.
      // if we get in here we can safely assume that el looks like this
      // { nextSibling: { parentNode: truthy } }
    }
    if(next.parentNode && next.nextSibling) {
      // next.parentNode == truthy
      // next.nextSibling == truthy

      // Like the previous example we can see that next has to have the two
      // properties; parentNode and nextSibling so el must look like this
      // { nextSibling: { parentNode: truthy, nextSibling: truthy } }
    }

    // if we are assuming at this point that next is an element, this next
    // statement will execute without a problem, because if it is a nextSibling
    // of the element it must have a parent. This is an example of where it may
    // be useful to define the relationship between the properties of an object
    // e.g. an Element cannot have a nextSibling/previousSibling unless it has a parent.

    // However we can get to this point with next being just any truthy value.
    // Now, it is entirely possible for next to have a property parentNode
    // with a function removeChild, but we can cause an error here using a number of
    // options. If parentNode is null or undefined, or if parentNode
    // doesn't have a function: removeChild
    // at this point, we know what can be passed in as an argument to cause an error
    // because we have built it up as we travel along.
    next.parentNode.removeChild(next);
  }
  // because throughout the condition statements we have been building a picture of
  // what each value “looks like”, we can tell what kind of value the result of the
  // function can be. This is useful because, even though javascript is a dynamically
  // typed language, we can tell what type the return value will be.
  return el;
}
```

Taking our example from the research report, this is how we would analyze it to find errors. The loop makes things much more complicated than the previous example.

```
function highlightNthSibling(el, n) {
  // el == anything
  // n == anything

  var i = 0;

  while(el && i++ < n) {
    // On the first iteration we know that el is at least truthy
    // el == truthy
    // i == 1
    // On subsequent iterations we can see that el becomes more complex
    // { nextSibling: { nextSibling: { nextSibling: .... } } }
    // i > 1

    // i - 1 < n
    el = el.nextSibling;
    // el == anything
  }

  // (el == truthy && i - 1 >= n) ||
  // (el == falsey && i - 1 < n) ||
  // (el == falsey && i - -1 >= n)
  // or just !(el == truthy && i - 1 < n)
  // i > 0

  if(el && i == n + 1) {
    // el == truthy
    // i == n + 1

    // An error can be caused here because
    // el can be any truthy value and
    // it cannot be guaranteed that it
    // will have a property called classList
    el.classList.add("highlight");
  }
}
```

This example is our fixed version of the last example. el is required to be an Element, this helps narrow down what kind of objects are allowed. One of javascript's useful features is that it is dynamically typed, so I feel we need to allow some leeway in what we consider an error.
For this reason in the implementation it may be useful to ask the user what types the inputs should be limited to.

```
function highlightNthSibling(el, n) {
  // el == anything
  // n == anything

  if(!el instanceof Element) return;
  // el instanceof Element
  var i = 0;

  while(el && i++ < n) {
    // el == truthy
    // i - 1 < n
    // We know that from the DOM specification, and our definition above, that
    // nextElementSibling will always be an Element or null.
    el = el.nextElementSibling;
    // el instanceof Element || el == null
  }

  // (el instanceof Element && i - 1 >= n) ||
  // (el == null && i - 1 < n) ||
  // (el == null && i - -1 >= n)
  // or just !(el instanceof Element && i - 1 < n)
  // i > 0

  if(el && i == n + 1) {
    // el instanceof Element
    // i == n + 1

    // This will no longer cause an error because an Element will always
    // have “classList” as a property, which in turn will always have “add” as a function
    el.classList.add("highlight");
  }
}
```

Bibliography

1. https://developer.mozilla.org/en-US/docs/JavaScript [Accessed 28th November 2012]
2. https://developer.mozilla.org/en-US/docs/Rhino [Accessed December 14th]
3. http://nodejs.org/ [Accessed 28th November 2012]
4. http://phantomjs.org/ [Accessed 28th November 2012]
5. http://www.jshint.com/ [Accessed 28th November 2012]
6. http://www.jslint.com/ [Accessed 28th November 2012]
7. **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
8. **C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert.** *Rozzle: De-cloaking internet malware***.** In IEEE Symposium on Security and Privacy, May 2012
9. http://jquery.com/ [Accessed 29th November 2012]
10. http://documentcloud.github.com/backbone/ [Accessed 29th November 2012]
11. http://documentcloud.github.com/underscore/ [Accessed 29th November 2012]
12. http://esprima.org/ [Accessed November 27th 2012]