Institiúid Teicneolaíochta Cheatharlach

INSTITUTE of TECHNOLOGY CARLOW

At the Heart of South Leinster

Project Report

17/04/2013

Daniel Connor
C00137906

Table of Contents

Introduction

In writing JSSeek we set out to see whether it would be possible to build a symbolic execution engine for javascript that would find statements where errors can occur. We highlighted that the most common type of error in Javascript is the type error which is thrown when an operation is performed on a variable whose type does not support the operation. This happens because Javascript is a dynamically typed language and variables can contain any of the 6 different types in Javascript.

There are many approaches to patching Javascript and some of the problems it has. Some of those being TypeScript[1] and GWT[2], which both attempt to bring some kind of static typing to Javascript by adding features to the language. They are then compiled to Javascript so that it can be run in the browser. While JSSeek sets out to solve a similar problem, it does it in a different way.

JSSeek uses symbolic execution which is the execution of code using symbolic inputs rather than real world ones. As a result of Javascript being dynamically typed, it is very hard to do symbolic execution. One of the benefits of using symbolic execution is that it is possible to create a specification for a function which defines the input and return types, then because JSSeek covers all paths in a function we can easily tell all of the return types that are possible. This is useful because a developer can then be sure that the function will always return a specific type such as in a statically typed language while still having the benefits of dynamic typing.

Other tools

Symbolic Execution

There are a some other tools that use symbolic execution to find security vulnerabilities or errors in Javascript[5, 6], however the majority of them limit the types of inputs to a subset of all Javascript types. Kudzu[6], which uses symbolic execution to find security vulnerabilities, allows strings integers and booleans as inputs but does not allow for symbolic objects. It mostly focuses on string inputs. Rozzle[5] also uses symbolic execution to find security vulnerabilities but instead of using symbolic inputs uses a tree of possible concrete values.

Random Testing

Other tools use random testing, which proves to be quite effective seeing as errors are mostly type errors and are caused by null or undefined values. As a result a lot of errors can be found by just using null or undefined as input values. Random testing is not very good however where there are more complex constraints in a function or if complex objects are required as inputs. In

these situations a guided approach such as symbolic execution is more useful.

At the moment javascript is the only language supported by the major browsers, as a result, to help solve some of the problems with Javascript is to create a new superset or completely different language that compiles to Javascript, which in my opinion is an inefficient way to do things. Now many changes are coming to Javascript as part of upcoming standards that solve a lot of the problems that developers complain about in the language. These changes involve introducing classes and block scoping for example. Javascript will however remain a dynamically typed language. This means that even though classes will be introduced, the properties of any object can still be changed, and variables will function in the same way as before. As a result JSSeek would be an important tool even in the foreseeable future of Javascript.

Another benefit of symbolic execution is that it is possible to create a specification for a function which defines the input and return types, then because JSSeek covers all paths in a function we can easily tell all of the return types that can be returned. This is useful because a developer can then be sure that the function will always return a specific type such as in a statically typed language while still having the benefits of dynamic typing.

Problems to solve

To find type errors there are four types of value that we need to look out for. Null, Undefined, Primitive types, and functions. Null and Undefined are two types that only have one value each. The problem with these types is that they don't allow property access and throw an error when attempted.

Primitive types, while property access is allowed through the prototype, they do not allow property assignment, however, when attempted this does not throw an error but fails silently which may cause errors later. This problem is expanded on in **Example D** below.

Functions in Javascript are first class objects, which means even though they can be invoked, they act like objects in every other way. However they are the only type of object that can be invoked, so when an attempt to invoke a normal object is made, an error is thrown.

Example A

This is the simplest example the error generator should be able to find an error in. If the obj is null or undefined, trying to access a property will throw a TypeError.

```
function example(obj, a) {
  if(obj[a]) {
    // do something
  }
```

```
}
```

JSSeek output

```
error:2:5 obj can be null or undefined
```

Example B

Objects in javascript are often used as namespaces to store configuration values and functions for easy access. In this example, even though the obj is defined within the function(I cannot currently handle global objects), a property name is passed as an argument which specifies a property of obj which should be called as a function. obj contains properties that are obviously not functions. When a value for prop is passed that does not correspond to a function, a TypeError will be thrown.

```
function example(prop, arg) {
  var obj = {
    a: {},
    b: {},
    c: null
  };

  if(typeof obj[prop] == “object”) {
    obj[prop].param = arg;
  }
}
```

JSSeek Output

```
error:8:4 undefined can be null or undefined
null, null
```

Example C

There are multiple different types in javascript(object, function, number, string, boolean etc.), which are further divided into two types; Primitive and reference. The main difference between them is that properties cannot be dynamically be added to primitives(they can be added to their prototypes which will then be accessible from instances of that type), and will fail silently if attempted. Now because it fails silently, this obviously won’t cause an error. However if you attempt to assign a property to a primitive value, then later try to access that property, it will not be there and the expression will give undefined. Now, an error will occur if you try to manipulate undefined, which won’t be the value you expected. An example of this is shown here. obj is

checked for the property “prop”, then if it doesn’t exist, it is added. Later in the function, the property “a” of the newly assigned property “prop” is assigned the value 10. If the input value obj is any primitive value, a TypeError will be thrown.

```
function example(obj) {
  if(obj != null) {
    if(obj.prop == null) {
      obj.prop = {};
    }
    obj.prop.a = 10;
  }
}
```

JSSeek Output

```
error:6:4 obj can be a primitive
52254
-53118
```

Example D

Objects in javascript have keys which are strings. Properties can be added dynamically to objects at runtime using any value as a key, however because keys must be strings, values other than strings must be converted to strings so they can be used as keys. Each type in javascript has a toString function on its prototype that gives the ability to convert a value to its string representation. In the case of numbers, the result is as you would expect. e.g `(1).toString() == "1"`. The same applies for booleans, and in the case of strings, the value is the same as that of the string. However in the case of complex types the result is not so obvious. Objects return `"[object Object]"` and Arrays return `"[object Array]"`. This means that any two different objects used as keys on another object will reference the same value of that object.

For example take two objects `a` and `b` which have different properties:
`var a = { a: 1 }, b = { b: 1 };`
Then take an object c:
var c = {};
Then use `a` as a key to add the value 1 to `c`:
`c[a] = 1;`
Now, access a property of `c` using `b` as a key which will give the value `1`.
`c[b] === 1; // true`
This happens because the string representation of objects `a` and `b` are used which are the

same. The object `c` looks like this:

```
{
  "[object Object]": 1
}
```

This example shows how this can cause an error when the properties aren't expected to be the same.

```
function example(obj, a, b) {
 if(obj != null && a !== b) {

   if(obj[b] == null) {
     obj[b] = {};
   }

   if(obj[a] != null) {
     obj[a] = null;
   }

   return obj[b].property = a;
 }
}
```

JSSeek output:

```
error:12:4 null can be null or undefined
{"null":-18386}, null, null
{"null":null}, null, null
error:12:4 obj can be a primitive
-26780, null, undefined
-9136, null, undefined
-64188, null, null
-39051, null, undefined
39475, null, null
6263, null, undefined
```

Design

Javascript

Firstly javascript is a dynamically typed language by design. There is lots of controversy over whether static or dynamically typed programming languages are better[7, 8]. Due to the fact

functions cannot declare the type of variables they take as arguments, or their return types javascript developers must check whether a function is called from a particular location. This can lead to very verbose and overly complicated code. One of the nice things about this, however, is that it's easier to create a function that takes different types and deals with them automatically rather than having to explicitly declare the arguments it takes. Once the function is written the developer should be able to expect that the function handles incorrect inputs gracefully without causing an error. This is the first problem that JSSeek sets out to solve. Basically what we try to do is verify that a function will never cause an error no matter the input.

States

As I set out in my design document, I planned to analyse each javascript statement separately using states to represent the values of variables in a particular path. When an if condition was encountered the current states would be "culled" to a particular subset of states which meet the constraints. The constraint imposed by the if statement is then stored on each state that does meet the requirements. The constraints represent everything that is known to be true about a particular state, therefore the entire constraint is a conjunction of all the constraints. As a result any OR statements/disjunctions are represented by creating a new state. This helps make things simpler when dealing with objects because when we are dealing with object properties, it is hard to represent that an object has one property, but doesn't have another without a separate state.

Unknowns

Unknown[1] variables represent input values, or members of unknown objects. They represent the multiple types that a variable can be at any one time. I decided that this would be the best way to represent multiple types rather than creating multiple states to represent each of the types as this would be very complex. There are 6 types in Javascript, so if I were to create a state for all the combinations of types of inputs there would be 6^n states, where n is the number of input variables. This would get even more complex when dealing with members of objects as one member would introduce another 6 states. Rather than doing this, I decided to associate types with each unknown variable. The main downside of this is that we lose any relationship between two variables and which values cause a constraint to be true. For example in the following expression, if we were to create a state for each of the types that a and b can be, then it would be represented something like this:

```
a - b == 1
```

a	b
true	false

[1] Previously known as VagueValues

`true`	`null`
`true`	`0`
`1`	`false`
`1`	`0`
1	null

However if we just use one state to represent an unknown, then the state in which that expression would be true is:
a: true, 1
b: false, null, 0

Checking Feasibility

As I've stated above a state must be checked for feasibility after each constraint is added. I've also stated that each Unknown represents the state of all the possible types of a variable. To check feasibility, JSSeek uses two approaches to narrow down the types that an unknown can be.

Type Inference

Type inferencing is often used by Javascript engines[4] to deduce the types of variables at particular locations in code. This allows them to be able to optimise the code based on specific types. While these systems often assume the minimum possible types, and then once it is found that there are other types that are possible, they are added. What we want to do is the opposite. We assume that a variable can be any possible type, and then based on constraints we get rid of types that are not longer possible.
Firstly, when a constraint is added to a state we can often check whether or not the types of a constraint allow the expression to be true. For example if the constraint is a != null. Then straight away we can rule out two possible types; Null and Undefined. This means that for the next step we have less types to check.

JSSeek currently support type inferencing for only some of the operators; equality operators, comparative operators, and truthy and not operators.

There are some cases where we can further optimise the constraints using type inference. For example if we have an equality expression involving an unknown and `null` or `undefined`.

Because the Null and Undefined types only have one value each, and they are equal to each other. Once we have inferred that the unknown is Null or Undefined, then no value of any other type can ever cause the expression to be true; nothing further can be deduced from this constraint. As a result we can ignore this constraint when brute forcing different types.

Brute Forcing

Due to the fact that the range of possible numbers is so large to check whether the constraints on a state are feasible, we use a constraint solver that can check this and also give an input that will pass the constraints. The solver we are using is ptc-solver. While in the first step we are able to reduce the number of types that an unknown can be, we cannot do this when constraints are made up of complex expressions where unknowns may be coerced into different types. To handle this we essentially brute force all the remaining types of the unknowns. For example, if there are two unknowns in a particular state; A and B. A can be a Boolean or a Number while B can be Null or Undefined. We need to try each combination of types in each of the constraints. We have a constraint: `A > 10 - B`

A	B	Description
Boolean	Null	Null is coerced to `0`, so `10 - B = 10`. `true` or `false` can be coerced to `0` or `1`
Number	Null	Null is again coerced to 0. There are no other constraints on A, so it is possible for A to be a Number type.
Boolean	Undefined	10 - B results in `NaN`, because B is undefined. There is no number greater than `NaN` so B cannot be Undefined.
Number	Undefined	The same result as previously.

We can now tell that A is a Boolean type and B is a Null type.

Generating inputs

Because of the way we store unknown values, when generating inputs that satisfy the constraints on an object, we check for feasibility as usual, but the first set of values that causes the constraint to be true can be used as inputs.

Problems Encountered

Objects/Hash Maps

Objects in Javascript are for the most part hash-maps, however they do provide more functionality to allow for prototypal inheritance and built-in functions which I will explain later. For now, we will look at how to represent the objects symbolically and the problems associated with that. The problem with objects acting like hash-maps is one of the problems we are trying to solve with Javascript, because we can never be sure that an object has a particular property or not, due to the fact that they can be added deleted or modified at will; as with any hash-map. There are two types of objects that we need to handle; concrete objects where we know what

properties it has, and unknown objects where the properties are unknown. Unknown objects are represented along with the other type information of unknown variables for reasons that we will explain later. There is one other difference between unknown objects and concrete objects but for now we can assume that they are represented in the same way.

Initially when an object was accessed using an unknown variable as a key, I tried to represent the resulting property using another unknown variable. It soon became apparent that this was quite a naive approach because it was impossible to represent the fact that the result could be any of the other properties that may have been assigned to it's container object previously.

When an unknown value is used as a property on either type of object, we create multiple states; for each key already in the object, a state is created where the unknown property is equal to the key, and another state where the unknown property is not equal to any of the keys in the object. In the case where the object is a concrete object the resulting value is undefined. Otherwise in the case where the object is an unknown object we return an new unknown value.

In this way multiple keys may be used to represent one property. We can keep track of these keys so that later in the code when a key is reused on an object on which it has been used before we can simply look it up using the same value without having to create new states. Again this is to help limit the amount of states that we have to keep track of states/branches we have to follow.

Performance

Performance is one of the main problems with symbolic execution, because as we have stated before, the number of branches/paths through the code becomes very large very quickly especially when dealing with weak typing. When creating the tool, I tried to minimize the memory requirements for keeping track of each states throughout the execution of the program. For the tool to be a useful one where it can be used, for example in an editor or IDE where the tool must be run as the user makes edits, it must run in a reasonable amount of time. As part of a build process this is not as big a concern however, it must obviously run in a reasonable amount of time. There is a large opportunity for parallelisation on multi-core machines due to the fact that each state can be analysed individually.

The tool is designed to analyse an individual function at any one time. It would not be possible to analyse a program all at once because the number of paths through the code grow exponentially as constraints are added. At each if statement, every state that is currently active must be split in two, one where the condition is met, and another where it is not. This could be handled more efficiently by using a more conventional method of analysing the code; that is handling each branch individually. JSSeek, in its current state, is not very memory efficient, as it keeps track of each state that can exist in the program, so when the number of branches gets large, the

amount of memory grows at the same rate. If this were to be done by analysing each branch individually, this problem would not exist as a branch would be followed to the end, or to a point where it no longer becomes feasible. This is essentially a depth first search approach rather than a breadth first search as JSSeek currently does.

Usable Tool

JSSeek as it stands is not a greatly usable tool. Along with supporting more Javascript features there are a number of things that prevent it from being a usable tool.

Analysing branches

As I've explained earlier one of the main problems with symbolic execution and Javascript is trying to represent all the types that a variable can be at once. One of the ways to solve this is by creating a branch for every combination of types of the inputs. The number of branches obviously becomes very large very quickly. The other way to solve this is by representing all of the possible types in one container which is what JSSeek does. With the first approach, because each of the unknowns in a branch only represents one type the entire branch becomes infeasible when it a certain constraint is applied.

In traditional symbolic execution each branch is analysed individually until it no longer becomes feasible or the end of the path is reached. Once this happens the analyser backtracks to the last branching point and continues analyses from there. In contrast to this, JSSeek executes each statement with all of the current states. If a statement causes new states to be created, these are added to the current list of states. The next statement is then executed with each of the states individually. The main problem with this is that the solver can only keep track of one state at a time. So when each state needs to be checked for feasibility, the solver must be reset and all of the constraints on the state submitted again. As well as that, each constraint must be submitted to the solver multiple times with each of the types that the unknowns can be.

A better approach would be a combination of both approaches described. An unknown should represent all the possible types that it can be at a particular stage of execution, but each branch should be analysed individually instead of keeping track of both states. Along with this a different solver would also be useful.

Javascript Subset

At the outset of this project I had planned to include a much larger subset of Javascript than I actually achieved. In this section I will set out the actual javascript subset that is handled by JSSeek and how it was achieved.

Logical Expressions

Logical expressions in Javascript are lazy evaluated like many other C-like languages and so can cause two different states to come about as the right hand expression is not always

evaluated. Also the result of the expression is not true or false as you would expect from other languages. In the case of the OR operator; the result is the result or the left expression if it is a truthy value, otherwise it is the result of the right expression. In the case of the AND operator; the result is the result of the left expression if it is a falsey value, otherwise it is the result of the right expression.

Due to the fact that javascript does not yet support default argument values, this mechanism is often used to give arguments default values. Here is an example:

```
function example(a) {
  a = a || true;
}
```

If the input “a” is any falsey value, then it will get assigned `true.` As you can see this acts like an if statement; an alternative, more verbose example, could look like this:

```
function example(a) {
  if(!a) {
    a = true;
  }
}
```

As a result the way that logical expressions are handled is very similar to the way if statements are.

Expressions

Expressions are represented as symbolic objects as they cannot be evaluated immediately. Expressions whose operands are all constant values are evaluated as soon as possible. For this reason, constraints including these types of expressions will be shown using the evaluated version of the expression.

While/Do While/For Loops

for(expression; expression; expression) statement
while(expression) statement

For loops are not supported by JSSeek at the moment, in any form.

For-In Loops

for(identifier in expression) statement

```
for(var i in obj) {
}
```

I had planned on the inclusion of for-in loop handling, and while I partially implemented the algorithm to handle them, I did not have time to finish it.
For-in loops in javascript allow the looping over properties of objects. This becomes difficult when dealing with unknown objects or concrete objects with keys that are unknown. To handle unknown objects within for loops, we just need to create two states. We first execute the loop body using every key that we know is on the object. Then we create another state an execute the body with a new unknown key.

Once a for in loop has been performed on an object, we have iterated over *all* of the properties in the object(Unless of course a break statement has been used to break out of the loop which we won't handle for now). For this reason we can treat the unknown object as if it were a concrete object from then on. Here is an example to demonstrate this:

```
function example(obj, prop) {
  for(var i in obj) {
      if(obj[i] % 2 = 0) {
          obj[i] = true;
      }
      else {
         obj[i] = false;
      }
   }
   if(obj[prop]) {
     return true;
   }
   else return false;
}
```

Because obj has not been accessed before the loop, we have no knowledge of any of the properties in obj. According to our algorithm above, we only need to create one new state where i is not equal to any of the keys in `obj`, of which there are currently none. Continuing into the for loop our state gets split into two. One where `obj[i]` is divisible by 2 and the other where it is not.

When we access `obj[prop]` then there is one state where prop is equal to i and another where `prop != i`. As we have covered all of the properties in obj, the case where prop != i, when we do a property access on `obj[prop]`, it should return `undefined`, because otherwise it will be `true` or `false`.

Functions

function identifier(argument list) { statements }

Functions are not currently handled by JSSeek, but they should not pose a problem to implement in the future. As we are only concerned with analysing each function individually, it shouldn't be a problem with complexity. The best way to handle functions would be to handle ones that are defined within another function scope.

```
function example(obj) {
  function internal(k) {...}
  for(var key in obj) {
    internal(key);
  }
}
```

As you can see the internal function will always be called as long as obj has any properties. The function is merely creating a new scope within the loop, so it must be analysed. We also know in this case that the input passed to internal will always be a string and so would limit the errors we need to check for.

It is often common to write javascript within an immediately invoked function expression to create a new scope and protect variables from the global scope. If this is the case then only analysing functions within other functions would still include the whole program.

```
(function() {
  var variable = 1;
  function things() {
  }
})();
```

Another option might be to just analyse the functions that get exposed to the global scope through some method. For example if a library were to expose functions like this:

```
window.DrawLib = (function() {
  function distance(pt1, pt2) {...}
  return {
    drawLine: function(pt1, pt2) {...},
    drawCircle: function(pt, radius) {...}
  };
})();
```

In this case the functions to be analysed would be anything in window.DrawLib. i.e. drawLine and

drawCircle.

Also in the case of the function distance, it would be beneficial to create a representation of the function that shows what paths are taken for a set of inputs, and whether an error will occur on that path. Then when the function is encountered, we can use the resulting value from the state where the inputs satisfy the constraints for that state. This would prevent having to symbolically execute the function each time it is encountered.

Strings

Because of Javascripts dynamic typing type coercion occurs quite frequently, for example, when an object is used in an arithmetic expression it gets coerced to a string. This is fine for plain objects which return a static string for the operation, but for instances of objects that override the default toString function, it's a different situation. The Array object when converted to a string concatenates its members separated by a comma. Without a proper string constraint solver it's impossible to handle any string modifications, no matter how simple.

The best option would have been to use a string solver such as Hampi[10] which I originally chose not to use because it would be more difficult to integrate due to the special syntax it uses to represent constraints. However it does use an SMT solver which could also have been used for number constraints. These solvers would be suitable assuming they could solve constraints between numbers and strings. Consider this constraint for example:

```
a < 100 && a > 0 && b.length == 2 && b == “” + a
```

a must be less than 5, greater than 0, and when converted to a string should be equal to a value b. One way of making this kind of thing simpler would be to simplify abstract operations, such as toString, to simple algorithms.

Numbers

Constraints involving numbers are handled through ptc-solver[ref], and all other numbers are handled as normal. The same problem applies to numbers as strings, because

Booleans

Constraints involving boolean values could be handled by ptc-solver[ref] but because it does not cast values the same way as Javascript. For this reason, because there are only two possible values, we use the brute force technique to check whether boolean types are feasible when checking constraints.

Prototypes/Constructors/instanceof operator

Prototypes and constructors are not currently supported by JSSeek. The instanceof operator

allows a programmer check whether an object in Javascript has a particular function as an instance of a constructor somewhere in its prototype chain.e.g.

```
function Constructor() {}
Constructor.prototype.func = function() {};
var a = new Constructor();
a instanceof Constructor === true
```

It helps to identify whether a particular object has properties that it inherits from its parents prototype. It would be useful to have in JSSeek, but it does not always guarantee that an object has a property. As a result, we represent objects as the minimum possible and only once it is proven that a property does exist can we say that an error will definitely not exist.

`this` variable

The `this`[9] keyword is not currently supported as JSSeek only runs in the scope of a single function and does not have any knowledge of how scope or context works in javascript.

Built-in functions

Almost every environment that Javascript runs in provides different built-in functions that are not implemented in Javascript. Rather, they are implemented in native code which cannot be analysed by the solver due to the fact that it is compiled. Originally I had planned to handle this kind of thing by creating a representation of the function that represents the paths that the function takes and the errors that occur on those branches.

For concrete values, because JSSeek is written in Javascript, we have access to the builtin functions that allow us to perform these operations directly on them. Also because of the way we are checking for feasibility we can use the builtin functions on constant values that we use such as null and undefined for example when converting them to strings. However in the case of numbers and strings we cannot do the same because they amount of values available for them is too large.

Learning Outcomes

Building and creating JSSeek was an interesting project and I learned a lot from it. It took me quite a long time to fully understand symbolic execution. I would say that this is partially because during the research stage I spent too much time trying to figure out problems to find in Javascript. Then once I had identified a problem, I did not do enough research into solutions to such problems. This led to further problems during the project when I had done some of my implementation and I discovered that that my solution for handling objects would not work. I then had to go back to the drawing board and come up with a better solution. This took quite a large amount of time as I had to re-implement a lot of code and I ended up not being able to include as large a subset as I wanted to.

For JSSeek I used Javascript as an implementation language. The reasoning for this was that it would make it more accessible to Javascript developers and a research project that is not easily integrated into any testing frameworks for Javascript. It also helped as I didn't need to define the semantics for handling concrete Javascript values. I don't think it was a bad choice, but I do think there are more suitable languages. One way of keeping it accessible to Javascript would be to write it in another language and create a binding like the one that I created for eclipse-clp[11].

I would also have liked to get more testing done on JSSeek, however due to time constraints I did not. As a result when I was doing some final testing I discovered a bug where I was getting inconsistent results depending on the order of a set of constraints. I think that with more strict test cases this error would have come to light earlier and saved me some time.

Bibliography

1. http://www.typescriptlang.org/ [Accessed 11/04/2013]

2. https://developers.google.com/web-toolkit/ [Accessed 11/04/2013]

3. **Avik Chaudhuri, Jeffrey S. Foster.** *Symbolic security analysis of ruby-on-rails web applications*. CCS '10 Proceedings of the 17th ACM conference on Computer and communications security, pages 585-594, 2010

4. **Hackett , Brian and Guo, Shu-yu**. *Fast and Precise Hybrid Type Inference for JavaScript*. PLDI '12 Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, pages 239-250, 2012

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

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. **Erik Meijer and Peter Drayton**. *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*. http://research.microsoft.com/en-us/um/people/emeijer/papers/rdl04meijer.pdf [Accessed 16/04/2013]

8. **Paulson, L.D**, *Developers Shift to Dynamic Programming Languages.* In Computer, Vol 40, Issue 2. pages 12-15, 2007

9. http://es5.github.io/#x11.1.1 [Accessed 16/04/2013]

10. **Adam Kiezun and Vijay Ganesh.** Proceeding ISSTA '09 Proceedings of the eighteenth international symposium on Software testing and analysis, pages 105-116

11. http://eclipseclp.org/ [Accessed 17/04/2013]

12. http://tc39wiki.calculist.org/es6/ [Accessed 17/04/2013]