Institiúid Teicneolaíochta Cheatharlach

INSTITUTE of TECHNOLOGY CARLOW

At the Heart of South Leinster

User Manual

17/04/2013

Daniel Connor
C00137906

Table of Contents

Introduction

JSSeek is a tool designed to find errors in Javascript code using symbolic execution. It analyses a function and if errors are found, sample inputs are given to show the error. It uses esprima.js as a Javascript parser and ptc-solver as a number constraint solver. It also uses a binding to ECLiPSe-CLP from Javascript through a C++ interface.

Installation

To instal JSSeek, the 32-bit version of node.js must first be installed from the http://nodejs.org. The 32-bit version is required because it must be linked with ECLiPSe-CLP, the environment that ptc-solver runs in, which is included in node-eclipse-clp. To build the binding from javascript to C, Visual Studio is required. All other dependencies will be downloaded automatically.

Once you have extracted the contents of the source code, open a cmd window in the directory run: `npm install -g`.

You can install JSSeek from the npm directory by running: `npm install -g jsseek`

The `-g` flag specifies that jsseek should be installed globally in the path. If you do not want this to be the case it can be omitted.

Limitations

JSSeek has quite a few limitations as it is only a proof of concept and only covers a subset of Javascript.

Types

JSSeek supports the following types: Null, Undefined, Number, Boolean, Object. It does not support the String type as it does not have a solver that supports constraints. Constant strings can be used in conjunction with the typeof operator for example. e.g. `typeof obj == “object`

Object properties are can only be the string representation of any of the types supported by JSSeek.

Flow of Control Statements

The only flow of control statements that are supported are if and else statements.

Examples

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
```

Usage

JSSeek provides both an API and a command line interface. Both are described below.

API

To obtain access to the API interface use `var jsseek = require("jsseek");` once it has been installed using npm.

The interface only exposes one function at the moment:

#analyse

Analyse takes string input and returns an ast in the the format as specified by the Mozilla Parser API. https://developer.mozilla.org/en-US/docs/SpiderMonkey/Parser_API

On any node that has caused an error will be appended a property detailing the reason for the error and each of the states/branches that will cause the error. The data structure is detailed below:

```
Node: {
  ...,
  errors: {
    <ERROR>: {
      args: <String>,
      states: Array<States>
    }
  }
}
```

Here's a sample of how to use the API:

```
var jsseek = require("jsseek");
var func = "function(obj) { obj.prop = 10; }";
var ast = jsseek.analyse(func);
```

ERROR

`NULL_UNDEFINED` - Logged when a property access is attempted on a possibly null or undefined value.

`PRIMITIVE_PARENT` - Logged when an property is accessed on a value whose parent may be a primitive value. This would cause the property to never be assigned in the first place, meaning the first parent would be undefined.

Command Line Interface

The command line interface allows a user to run jsseek on a file. Any function in the root of the file will be analysed.

```
jsseek file
```

The format of output of JSSeek is as follows.

```
possible-error:<line_no>:<column> <description>
sample-arguments: <argument, ...>
```