Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Test Data Generation for JavaScript Functions that Interact with the DOM

Alexander Elyasov, Wishnu Prasetya, **Jurriaan Hage**

Utrecht University, The Netherlands
alex.elyasov@gmail.com

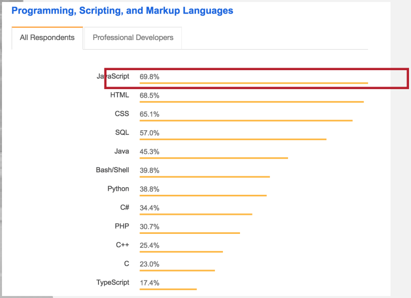
ISSRE '18, Memphis, TN, US

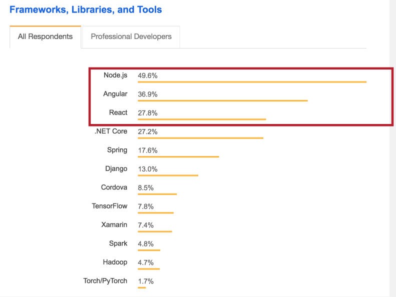
Outline

1. Introduction

StackOverflow Developer Survey

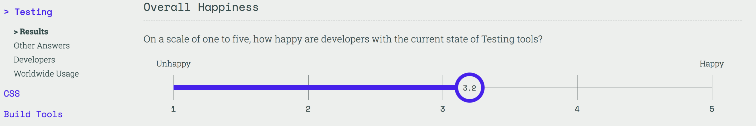
- For the sixth year in a row, JavaScript is the most commonly used programming language
- Node.js and AngularJS continue to be the most commonly used technologies, with React also important to many developers





The State of JavaScript Testing

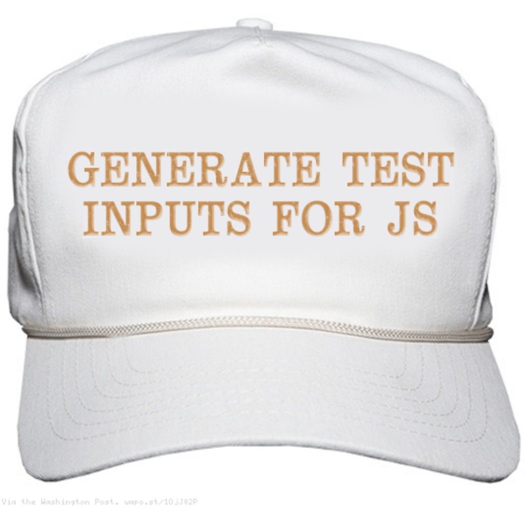
- There exist numerous JS frameworks on the market:
  - structure: Mocha, Jasmine
  - assertions: Chai
  - mocks and spies: Sinon
  - coverage: Istanbul
  - reporting: Karma
- Developers report a relatively low happiness score with the state of testing tools



Goal


MAKE JS TESTS
GREAT AGAIN
Via the Washington Post. wapo.st/1OjJ02P

Eureka!


GENERATE TEST
INPUTS FOR JS
Via the Washington Post. wapo.st/10jJ02P

Test Input Generation

Available techniques

- concolic execution: Jalangi and Confix
- random generation: JSContest
- static and dynamic analysis: Artemis
- crawling: Crawljax

Input types

- primitive types: numbers and strings
- collections: arrays
- objects
- DOM (focus of this paper)
- functions

Solution

JEDI: **J**avascript **E**volutionary testing framework with **D**OM as an **I**nput

Contributions

- implemented JEDI
- test generation algorithm
- evaluation

2. Motivation Example

Sudoku

6		2	9	5	8			7
8		1	4		7	5	2	3
			3	1	2	9		
2	1		7	9			8	5
9				3				6
7	5			8	1		9	2
		8	5	2	9			
3	2	5	1		6	8		9
1			8	7	3	2		4

isGameFinished

```
/*t dom */
function isGameFinished() {
  var obj = document.getElementById('sudoku');
  var subDivs = obj.getElementsByTagName('DIV');
  var allOk = true;
  for (var no = 0; no < subDivs.length; no++) {
    if (subDivs[no].className.indexOf('square') >= 0
        && !subDivs[no].style.backgroundColor) {
      var spans=subDivs[no].getElementsByTagName('SPAN');
      if (spans[0].innerHTML != spans[1].innerHTML) {
       allOk = false; //target
       break;
      }
    }
  }
  return allOk;
}
```

DOM Tests

```
1  <!-- T1: (5,5) -->
2  <html>
3   <body>
4    <div id='sudoku'>
5  
6    </div>
7   </body>
8  </html>
```

```
1  <!-- T2: (5,6), (7,15) -->
2  <html>
3   <body>
4    <div id='sudoku'>
5     <div></div>
6    </div>
7   </body>
8  </html>
```

```
9
10 <!-- T3: (7,8), (10,14) -->
11 <html>
12  <body>
13   <div id='sudoku'>
14    <div class='square'>
15     <span></span>
16     <span></span>
17    </div>
18   </div>
19  </body>
20 </html>
```

```
10 <!-- T4: (7,8), (10,11) -->
11 <html>
12  <body>
13   <div id='sudoku'>
14    <div class='square'>
15     <span></span>
16     <span>TEST</span>
17    </div>
18   </div>
19  </body>
20 </html>
```

DOM Tests

```
<!-- T1: (5,5) -->
<html>
 <body>
  <div id='sudoku'>

  </div>
 </body>
</html>
```

```
<!-- T2: (5,6), (7,15) -->
<html>
 <body>
  <div id='sudoku'>
   <div></div>
  </div>
 </body>
</html>
```

```
<!-- T3: (7,8), (10,14) -->
<html>
 <body>
  <div id='sudoku'>
   <div class='square'>
    <span></span>
    <span></span>
   </div>
  </div>
 </body>
</html>
```

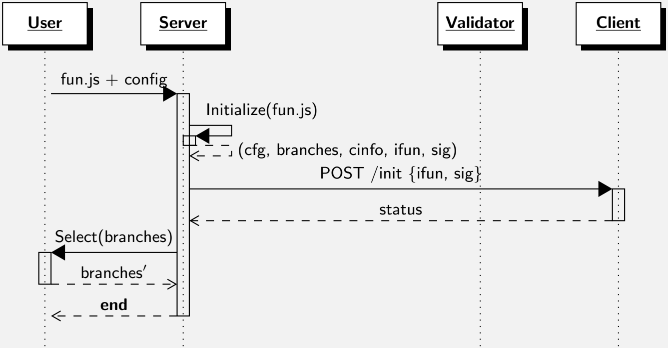
```
<!-- T4: (7,8), (10,11) -->
<html>
 <body>
  <div id='sudoku'>
   <div class='square'>
    <span></span>
    <span>TEST</span>
   </div>
  </div>
 </body>
</html>
```

T4 Explained

```
/*t dom */
function isGameFinished() {
  var obj = document.getElementById('sudoku');   // ☞ <div id='sudoku'>
  var subDivs = obj.getElementsByTagName('DIV'); // ☞ <div class='square'>
  var allOk = true;
  for (var no = 0; no < subDivs.length; no++) {
    if (subDivs[no].className.indexOf('square') >= 0  // ☞ class='square'
        && !subDivs[no].style.backgroundColor) {
      var spans=subDivs[no].getElementsByTagName('SPAN');//☞ <span></span>
      // ☞ <span>TEST</span>
      if (spans[0].innerHTML != spans[1].innerHTML) {
       allOk = false; //target
       break;
      }
    }
  }
  return allOk;
}
```

3. Test Generation Framework

Initialization Phase



Algorithm I: Initialization Phase

Input : JS file $fun.js$ with FUT and type annotation
Output : Tuple $(cfg, branches, cinfo, ifun, sig)$

Function $Initialize(fun.js)$
  $(ast, sig) \longleftarrow ParseFuncAndSig(fun.js)$
  $cinfo \longleftarrow GetConstantInfo(ast)$
  $nast \longleftarrow NormalizeAST(ast)$
  $ifun \longleftarrow Instrument(nast)$
  $cfg \longleftarrow BuildCFG(nast)$
  $branches \longleftarrow GetBranches(cfg)$
  return $(cfg, branches, cinfo, ifun, sig)$

Supported Types

Type Annotation

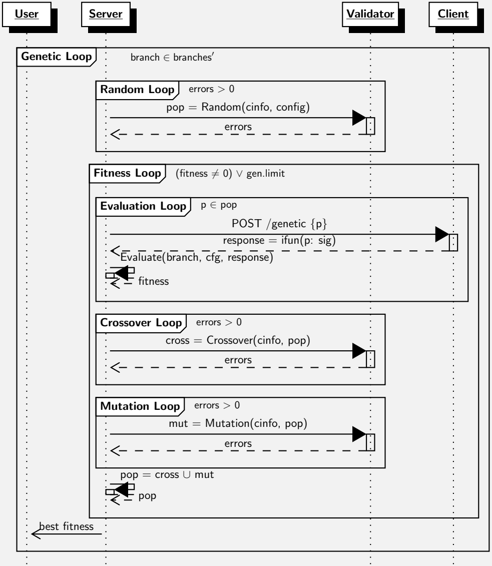
/*t dom : bool : int : float : string : [int] */

- primitive types: bool, int, float, sting
- arrays
- DOM
- could be extended to objects

Instrumentation

- trace records the sequence of executed statements of the FUT
- branchDistance contains the distance from the target branch
- loopMap captures the upper bound for the number for-loop iterations

Genetic Phase



HTML Generation

Random

- DSL for the generation of syntactically valid HTML documents composed of arbitrary tags and attributes
- multiple options for the parameterization: tree width and depth, tags frequency, etc.

GA operations

- crossover two HTML trees at a randomly chosen node
- HTML tree mutations: *NewTree*, *DropTree* and *ShuffleAttributes* (e.g. 'id' and 'class')

Fitness Function (FF)

FF for a node

$$F(n) = \mathit{approach\_level} + \begin{cases} 1/2 * (\frac{\mathit{branch\_distance}}{1+\mathit{branch\_distance}}) & \text{if no exception} \\ 1 & \text{otherwise} \end{cases}$$

- approach_level is the number of decision nodes between the target and the problem node (in JS every node can be exceptional)
- branch_distance measures the deviation explicitly in the problem node

FF for a node sequence

$$F^*(n_b, n_x) = (F(n_b), F(n_x))$$

- n_b is a target branch node
- n_x is a terminal exit node
- Can be generalized to: $F^*(n_1, n_2, \ldots, n_k) = (F(n_1), F(n_2), \ldots, F(n_k))$

GA with Restart

Problem

GA sometimes reaches a local minimum and stagnates due to flag variables, nested predicates, or unstructured control flow.

- Common solution it to incorporate data dependency into the search, e.g. chaining approach [FK96]
- Our approach is CFG-based because JS is a dynamic language which is hard to analyze for data dependency
- We suggest to restart GA with a new target when the fitness progress stops during a configured history window
- The new target consists of the original target prefixed by a dominating CFG node

4. Empirical Evaluation

Genetic Algorithm Configurations

Variations of Generation Algorithms

- $\mathbb{R}$ random
- $\mathbb{G}$ pure genetic
- $\mathbb{G}_{10}$ genetic with restart

Parameter Name	$\mathbb{R}$	$\mathbb{G}$	$\mathbb{G}_{10}$
Population size	50	50	50
Archive size	1	25	25
Maximum number of generations	200	200	200
Crossover rate	0	0.5	0.5
Mutation rate	0	0.5	0.5
History window size	-	-	10

Case Studies

case-study	function	loc	c	d	cc	dom	id	tag	class
sudoku	helpMe(int,int)	14	2	3	3	+	+	+	-
sudoku	isGameFinished()	10	3	3	4	+	+	+	+
sudoku	newGame()	9	1	2	2	+	+	+	+
sudoku	revealAll()	8	0	2	1	+	+	+	-
sudoku	shuffleBoard(int,int)	23	2	3	3	+	-	+	-
phormer	toggleInfo(string)	16	3	1	4	+	+	-	-
hotel RS	isValidCard([int])	17	2	2	5	-	-	-	-
hotel RS	isValidVISA([int])	6	3	1	6	-	-	-	-
apophis	initShields([int],int,int)	6	0	1	1	+	+	-	-
bingbong	brickJiggler(int,int,[int],[int],[int],[int])	7	1	1	2	+	+	-	-
bingbong	doPaddlePower(int,int)	15	2	1	3	+	+	-	-
bingbong	initBricks(int,[int],[int],[int],[int],int,[string])	70	12	4	13	+	+	-	-
burncanvas	do_draw(int,int,int,int,int,int,int)	40	12	2	14	-	-	-	-
mathjs	prob_gamma(float)	57	8	2	16	-	-	-	-

- if CC < 10 then the function is easy to test else difficult

Research Questions

RQ1 What is the branch coverage?

RQ2 What is the coverage time per branch?

RQ3 Are the results statistically significant?

RQ4 What is the branch coverage of Confix?

5. Results

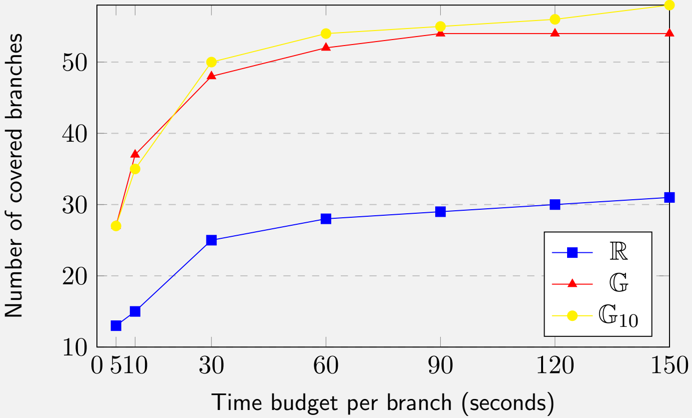
RQ1: What is the branch coverage?

Table: Branch coverage (%)

TYPE	$\mathbb{R}$	$\mathbb{G}$	$\mathbb{G}_{10}$
simple (23)	79	97	100
difficult (33)	58	94	100
global (56)	63	95	100

Across all subjects, the $\mathbb{G}_{10}$ algorithm achieved 100% branch coverage, with $\mathbb{G}$ in the second place with 95% coverage, and, finally, $\mathbb{R}$ with 63% coverage.

RQ1: Branch Coverage Progress



$\mathbb{G}$ could be more suitable for rapid testing during development, whereas $\mathbb{G}_{10}$ for integration

RQ2: What is the coverage time per branch?

Table: Average execution time per brunch (sec.)

TYPE	$\mathbb{R}$	$\mathbb{G}$	$\mathbb{G}_{10}$
simple (23)	37	9	7
difficult (33)	112	56	25
global (56)	88	39	19

$\mathbb{G}_{10}$ outperformed both $\mathbb{G}$ and $\mathbb{R}$ with an average execution time per branch of 19, 39 and 88 seconds, respectively.

On average, one algorithm iteration took one second for $\mathbb{R}$ and $\mathbb{G}_{10}$, $\mathbb{G}$ performed somewhat worse at 1.4 seconds.

RQ3: Are the results statistically significant?

TYPE	$\mathbb{R}/\mathbb{G}$			$\mathbb{R}/\mathbb{G}_{10}$			$\mathbb{G}/\mathbb{G}_{10}$		
	L	M	S	L	M	S	L	M	S
simple (23)	11	12	13	11	13	13	1	1	1
difficult (33)	22	24	26	22	23	26	2	2	4
global (56)	33	36	39	33	36	39	3	3	5

- Use the non-parametric Mann-Whitney U-test and the Vargha-Delaney $\hat{A}_{12}$ statistics for measuring statistical significance ($\alpha = 0.05$) and effect size [AB11]
- L - large (0.71), M - medium (0.64), S - small (0.56)

Both $\mathbb{G}$ and $\mathbb{G}_{10}$ largely outperform $\mathbb{R}$ in 50% cases generally across the board and 67% on the difficult functions.

RQ4: What is the branch coverage of Confix?

case-study	function	#BR	#C (weak)	#C (strong)	#tests	time (sec.)
sudoku	helpMe	5	4 (80%)	4 (80%)	2	5
sudoku	isGameFinished	5	2 (40%)	2 (40%)	2	5
sudoku	newGame	3	3 (100%)	3 (100%)	4	6
sudoku	revealAll	2	2 (100%)	0 (0%)	6	11
sudoku	shuffleBoard	7	5 (71%)	0 (0%)	2	5
phormer	toogleInfo	4	2 (50%)	2 (50%)	3	5
apophis	initShields	1	0 (0%)	0 (0%)	1	6
bingbong	brickJiggler	2	0 (0%)	0 (0%)	1	3
bingbong	doPaddlePower	4	2 (50%)	2 (50%)	1	4
bingbong	initBricks	18	3 (17%)	0 (0%)	1	3

The choice between a concolic and search-based approach is a trade-off between a labour-intensive modelling and execution time, respectively, where both can reinforce each other.

Threats to Validity

- construct validity: measured only branch coverage and execution time
  - it doesn't indicate fault finding capability
  - use only "natural" oracles such as exception
  - ignore test case size
- internal validity: choice of the initial GA configuration
- conclusion validity: stochastic nature of the experiment
- external validity: limited set of experimental subjects

6. Future Work and Conclusions

Future Work

- the whole test suite generation based on multi-objective optimization to balance coverage and test suite size [PKT17]
- mutation driven test generation [FZ12]; should allow us to evaluate fault finding capability
- combine search-based with concolic test generation for JavaScript, e.g. integrate Jalangi [SKBG13] and Confix [FMW15]
- conduct large evaluation of JEDI; it requires the extension of arbitrary input objects and functional arguments [CH11, SPRT18]

Conclusions

- introduced a novel search-based JS unit testing framework, called JEDI, which is able to generate arbitrary DOM inputs
- presented a test generation algorithm —"genetic with restart"; it is capable of escaping plateaus by concretizing the search target
- conduced an empirical validation followed by the significance study, which has confirmed the effectiveness and efficiency of our framework in branch coverage testing

References I

Andrea Arcuri and Lionel Briand.
A practical guide for using statistical tests to assess randomized algorithms in software engineering.
In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1–10. IEEE, 2011.

Koen Claessen and John Hughes.
Quickcheck: a lightweight tool for random testing of haskell programs.
Acm sigplan notices, 46(4):53–64, 2011.

Roger Ferguson and Bogdan Korel.
The chaining approach for software test data generation.
ACM Transactions on Software Engineering and Methodology (TOSEM), 5(1):63–86, 1996.

Amin Milani Fard, Ali Mesbah, and Eric Wohlstadter.
Generating fixtures for JavaScript unit testing.
In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 11 pages. IEEE Computer Society, 2015.

Gordon Fraser and Andreas Zeller.
Mutation-driven generation of unit tests and oracles.
IEEE Transactions on Software Engineering, 38(2):278–292, 2012.

Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella.
Lips vs mosa: a replicated empirical study on automated test case generation.
In *International Symposium on Search Based Software Engineering*, pages 83–98. Springer, 2017.

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs.
Jalangi: a selective record-replay and dynamic analysis framework for javascript.
In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.

References II

Marija Selakovic, Michael Pradel, KARIM REZWANA, and Frank Tip.
Test generation for higher-order functions in dynamic languages.
OOPSLA, 2018.

The End

Thanks for attention!

Questions?