Syntest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript

Olsthoorn, Mitchell; Stallenberg, D.M.; Panichella, A.

DOI
10.1145/3643659.3643928

Publication date
2024

Document Version
Final published version

Published in
2024 ACM/IEEE International Workshop on Search-Based and Fuzz Testing

Syntest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript

Mitchell Olsthoorn
Delft University of Technology
Delft, The Netherlands
M.J.G.Olsthoorn@tudelft.nl

Dimitri Stallenberg
Delft University of Technology
Delft, The Netherlands
D.M.Stallenberg@tudelft.nl

Annibale Panichella
Delft University of Technology
Delft, The Netherlands
A.Panichella@tudelft.nl

ABSTRACT

Over the last decades, various tools (*e.g., AUSTIN* and *EvoSuite*) have been developed to automate the process of unit-level test case generation. Most of these tools are designed for statically-typed languages, such as *C* and *Java*. However, as is shown in recent Stack Overflow developer surveys, the popularity of dynamically-typed languages, such as *JavaScript* and *Python*, has been increasing and is dominating the charts. Only recently, tools for automated test case generation of dynamically-typed languages have started to emerge (*e.g., Pynguin* for *Python*). However, to the best of our knowledge, there is no tool that focuses on automated test case generation for server-side *JavaScript*. To this aim, we introduce *SynTest-JavaScript*, a user-friendly tool for automated unit-level test case generation for (server-side) *JavaScript*. To showcase the effectiveness of *SynTest-JavaScript*, we empirically evaluate it on five large open-source *JavaScript* projects and one artificial one.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; **Software testing and debugging**.

KEYWORDS

software testing, search-based software testing, test case generation, fuzzing, javascript, syntest

ACM Reference Format:
Mitchell Olsthoorn, Dimitri Stallenberg, and Annibale Panichella. 2024. Syntest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript. In *2024 ACM/IEEE International Workshop on Search-Based and Fuzz Testing (SBFT '24), April 14, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3643659.3643928

1 INTRODUCTION

Software testing is an important part of the software development process. This task is often performed manually, which can be both time-consuming and prone to errors. To automate this process, various tools for unit-level test case generation (*e.g., AUSTIN* for *C*, and *EvoSuite* and *Randoop* for *Java*) have been created over the years. These tools mostly focus on statically-typed languages [2].

SBFT '24, April 14, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0562-5/24/04.
https://doi.org/10.1145/3643659.3643928

The most recent Stack Overflow developer survey[1], however, shows that *JavaScript* and *Python*, which are both dynamically-typed, are the most popular programming languages among professional developers. Recently, Lukasczyk and Fraser proposed *Pynguin*, an automated unit-level test case generation tool for *Python* [6]. However, despite *JavaScript*'s eleventh year in a row as the most popular programming language, automated tool support for test case generation for *JavaScript* is still lacking.

In the last decade, there has been a growing interest in developing tools for *JavaScript* [1, 5, 8, 9]. These tools, however, focus on *JavaScript* web applications that are characterized by their event-driven execution model and interaction with the Document Object Model (DOM) of the browser. *JavaScript* started out in 1995 as a client-side scripting engine for the browser, but through the years, additional *JavaScript* runtime engines like Node.js, Deno, and Bun have emerged, which allow developers to use *JavaScript* for server-side applications. These server-side *JavaScript* engines are used to create web servers and command-line tools and are heavily used by companies like Netflix[2], PayPal[3], and Uber[4].

A crucial problem with developing tools for dynamically-typed languages is that these types of languages do not provide any information on the types of variables and parameters. Types are instead inferred during the execution of the code. This characteristic, coupled with *JavaScript*'s weak typing —where variables can change types during execution— complicates the static determination of types. Without knowing the type of a function parameter, it will be challenging to generate the appropriate test inputs.

In this paper, we introduce *SynTest-JavaScript*, an open-source automated unit-level test case generation tool for *JavaScript*, which uses a probabilistic type inference approach we have introduced in our previous work [12]. It makes use of search-based algorithms to generate test cases that maximize function, branch, and path coverage. *SynTest-JavaScript* is implemented on top of the *SynTest-Framework*, which is a modular and extensible ecosystem for testing tools. This tool aims to provide a platform for researchers and practitioners to develop and evaluate new techniques for test case generation of *JavaScript* programs. A key feature of *SynTest-JavaScript* is its plugin-friendly architecture, which allows additional search algorithms and genetic operators to be easily added.

We performed an empirical study to evaluate the effectiveness (*i.e.,* branch coverage) of our tool for generating test cases for 99 *JavaScript* source code files. This evaluation shows that *SynTest-JavaScript* can on average, achieve 69.4 % of branch coverage with the state-of-the-art search algorithm *DynaMOSA* [11].

[1] https://survey.stackoverflow.co/2023/#most-popular-technologies-language-prof
[2] https://netflixtechblog.com/debugging-node-js-in-production-75901bb10f2d
[3] https://paypal.github.io/PayPal-node-SDK/
[4] https://www.uber.com/en-NL/blog/uber-tech-stack-part-two/

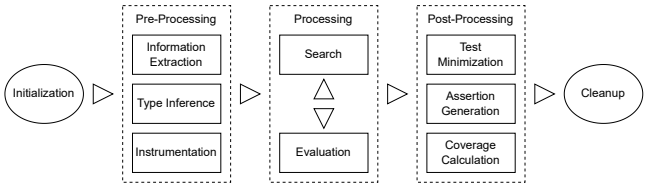

Figure 1: *SynTest-JavaScript* tool workflow

2 SYNTEST-JAVASCRIPT

SynTest-JavaScript is an automated unit-level test case generation tool for (server-side) *JavaScript* code within the *SynTest-Framework* ecosystem. Users can interact with the tool through the CLI of the *SynTest-Framework*. To run the tool the following command structure can be used "`syntest javascript <command> [options]`". For more information on how to run the tool and its options, see the documentation[5]. The tool can be found on GitHub[6]. In the following section, we will discuss the workflow and highlight the critical components of the tool.

2.1 Workflow

The workflow of our tool, depicted in Fig. 1, unfolds across five phases: (i) *initialization*, (ii) *pre-processing*, (iii) *processing*, (iv) *post-processing*, and (v) *cleanup*.

The *initialization phase* consists of setting up the environment, configuring all the required variables, and initializing the required classes. Next, the *pre-processing* phase uses static analysis methods to gather information about the targeted `units` (*i.e.,* exported functions or classes) that can be used to improve the search process. In this phase, we build the Control Flow Graph (CFG) starting from the Abstract Syntax Tree (AST) of the unit under test. The CFG allows us to extract the branch/function/path objectives from each `unit`. These objectives are used during the *processing* phase to guide the search algorithms towards maximum coverage. Next, we infer the variable types using the type inference techniques as proposed in our previous work [12]. Finally, we instrument the source code. This instrumentation allows us to record information about the performance of our generated test cases.

During the *processing* phase, each targeted file is considered separately. The information gathered in the pre-processing phase is used to sample encodings (test cases) during the search process. These encodings are then evaluated based on the distance from the objectives, which is calculated by executing the generated test cases (encodings) and using the coverage data generated by the instrumentation. For every objective that has been covered, we save an encoding in our archive [10]. Next to the original objectives (*e.g.,* branches), we also save error objectives that are discovered during the search process. The search and evaluation go back and forth until one of the stopping criteria is met (*e.g.,* running time).

In the *post-processing* phase, we optimize and prettify the encodings (test cases) in the archive. To achieve this, we first minimize the size of the test cases by iteratively removing spurious statements that do not contribute to the total coverage [11]. Next to the individual test case minimization, we also reduce the entire archive (test suite) by checking whether two test cases cover the same objectives and removing one of them. After minimization, the tool generates assertions for each function call result, or exception thrown. Finally, the resulting test suite is run to calculate the final coverage. An example of a generated test case with assertions is shown in Figure 3. In the last phase, the tool *cleans up* all the generated temporary files.

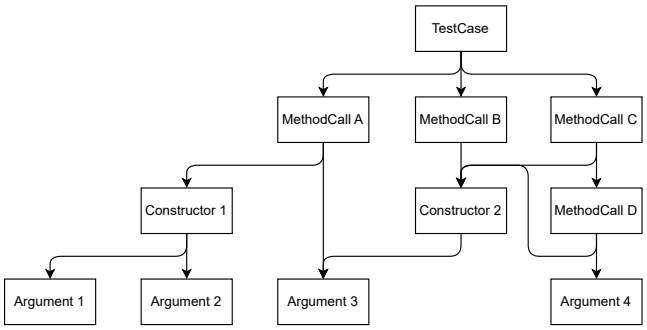

Figure 2: *SynTest-JavaScript* encoding structure

2.2 Components

Presets. Presets allow developers or researchers to create pre-specified configuration settings. Currently, we have four options, *random search*, *NSGAII* [4], *MOSA* [10], and *DynaMOSA* [11]. Each preset is designed to align with the configurations detailed in their respective original articles.

Encoding. Our encoding for test cases is structured as a directed acyclic graph. An example of such an encoding is shown in Fig. 2. At the top, we have the test case itself, which contains the root statements. In this example, the root statements consist of three method calls. Each method call requires an instance of an object to be called upon, for this reason, each method call has a constructor child. Next to the constructor, some method calls have arguments. These arguments can be primitives (*e.g.,* `boolean`), objects, functions, or results of other method calls. In Fig. 2, we see that method call C uses the result of method call D as an argument. Although not shown in the Fig. 2, the roots may also be object function calls or regular function calls, the regular function call does not require an object instance to be called.

Supported types. Our encoding supports two primitive types, namely complex and action statements. The primitive statements are a reflection of the primitive statements in *JavaScript* itself. These include: `boolean`, `integer`, `null`, `numeric`, `string`, and `undefined`. Note that in JavaScript there is no distinction between numeric and integer. However, to improve the capabilities of the tool we included a separate integer statement. Currently, the tool supports the following complex statements: `arrays`, `arrow functions`, and `objects`. Finally, the action statements include the `constructor`, `function`, and `method` calls. When the type matching engine finds a matching class type for a certain variable, we can import the matching class and instantiate it through a constructor call. If however no matching class can be found, we use the object statement

[5]https://www.syntest.org/docs/
[6]https://github.com/syntest-framework/

to construct the required type. This enables the tool to support an infinite number of types.

Constant Pool. During the static analysis in the pre-processing phase, we gathered all constant values from the source code and put them into a constant pool. These constants can then be used during the sampling of primitive types such as strings and numbers.

Type Pool. Next to the constant pool, we also create a type pool, using the analysis files, which consists of all the user-defined object types (classes, interfaces, prototyped functions, etc.). These types can then be used when certain objects need to be sampled. We try to find the most likely match to the required object and then sample a constructor or import of that type. As mentioned before, if no matching type can be found, the sampler constructs the object itself through an object statement.

Statement Pool. For each test case, we maintain a statement pool that consists of each statement within the encoding tree. During the sampling of new statements for a test case, there is a chance of reusing already occurring statements from the encoding tree. This is done by sampling a matching statement from the statement pool. For this reason, our encoding is a directed acyclic graph instead of a tree. Fig. 2 shows this by for example method calls B, C, and D all using constructor instance 2. Note that this only works when the types of the statements match.

Execution engine. To ensure that test case executions do not influence each other we created a test case execution engine that runs each test case in a new process. Running a test case in a separate process allows us to terminate the execution in case of a timeout or memory overflow (which can happen with generated test cases). The execution engine provides a separate process with the test to execute and the relevant environment. After execution, the results sent back by the process include instrumentation data, meta-data, and assertion data. To calculate the "fitness" of a test we measure its distance to cover all unreached branches in the code, as typically done in *DynaMOSA* [11]. The distance to each uncovered branch is computed using two well-known coverage heuristics [7]: (1) the *approach level* and (2) the normalized *branch distance*. We use the instrumentation data to calculate the approach level. To calculate the branch distance we use the meta-data which consists of the branch conditions together with the relevant variable values. Finally, the assertion data contains the results of function calls and is used to generate assertions.

Test splitting. As mentioned before, the post-processing phase minimizes the size of each test case by splitting them. Take the encoding shown in Fig. 2 as an illustrative example. In this scenario, the original test case can be split into two separate ones: the first encompassing method call A, along with its associated child statements; the second comprising methods calls B and C, along with their child statements. The tool then runs these two test cases separately and checks whether their combined coverage is equal to (or higher than) the original test. In that case, the two new tests are stored and further considered for additional splits recursively.

Test de-duplication. After the test splitting, we end up with a large set of test cases, some of which might be redundant w.r.t. to the final coverage. For this reason, we have a de-duplication step in our workflow. During this step, each test case is compared to the other test cases to check for duplicate objective coverage. If two test cases cover exactly the same objective, the best one is picked based on secondary objectives such as length or readability.

```
1  it("suggestSimilar returns correct suggestion for a misspelled
       word with special characters", async () => {
2    // Meta information
3    // Selected for objective: ./suggestSimilar.js
         :80:13:::82:7:::2311:2384
4    // ...
5    // Covers objective: ./suggestSimilar.js
         :81:8:::81:32:::2352:2376
6
7    // Test
8    const word = "cac@e-valiate";
9    const arrayElement = "cache-validate";
10   const candidates = [arrayElement]
11   const suggestSimilarReturnValue = await suggestSimilar(word,
         candidates)
12
13   // Assertions
14   expect(suggestSimilarReturnValue).to.equal("\n(Did you mean
         cache-validate?)")
15 })
```

Figure 3: Example of generated test case

Meta-commenting. To provide as much information as possible to the end user the tool provides meta-comments in each test case. These comments provide information about which objectives the test case covers and for which objective the test case was chosen. For error objectives, we also provide the stack trace in the comments. Fig. 3 shows some meta-comments in line 2 to 5.

Naming strategy. To generate test cases that not only achieve high coverage but are also very readable, the names of the used variable names must be logical. To achieve this, the tool uses the names of the parameters of the called functions as the variable names for the corresponding arguments. If a variable name is already in use, we number them. For return values, we currently simply name the variable `"[function name]ReturnValue"` as can be seen on line 11 in Fig. 3. In the future, we plan to improve this by using the name of the returned variable in the source code. We also plan to improve the test and variable names by using Large Language Models (LLMs) as a prettifier.

Assertion Generation. A test case is incomplete without proper assertions. To generate assertions we first execute the test cases without any assertions and record the result of each function call. In the case of an error, we catch and record the error. Then the recorded results are asserted in the final test suite. An example of this is shown on line 14 in Fig. 3.

3 EVALUATION

To evaluate the effectiveness of *SynTest-JavaScript*, we performed an experiment on the *SynTest-JavaScript-Benchmark*, previously introduced in [12]. To the best of our knowledge, this is the only benchmark targeted at unit-level test case generation for *JavaScript*. The current version of the benchmark contains 99 *JavaScript* source code files which consist of popular *JavaScript* libraries that represent a diverse set of *JavaScript* syntax and code styles. Table 1 provides the main characteristics of the benchmark projects, including the number of `files`, the number of `units` (*i.e.,* exported

Benchmark	Metrics			Achieved Branch Coverage			Statistical Significance		
	Files	#Units	Avg. CC	random	DynaMOSA	Difference	#Lose	#No Diff.	#Win
Artificial	4	4	5	47.92%	87.50%	39.58%	0	1	3
Commander.js	4	6	23	56.24%	75.43%	19.20%	0	0	4
Express	6	12	32	46.30%	46.41%	0.11%	2	3	1
JavaScript Algorithms	56	69	10	67.88%	73.67%	5.79%	3	28	25
Lodash	10	10	11	81.59%	89.13%	7.54%	0	7	3
Moment.js	19	41	18	45.39%	48.59%	3.20%	1	13	5
Average	17	24	17	57.55%	70.12%	12.57%	1	9	7

Table 1: Overview of the benchmark metrics, achieved coverage, and statistical significance

classes or top-level functions), and the average Cyclomatic Complexity per file (**CC** column).

We used the state-of-the-art search algorithm *DynaMOSA* [11] and compared it against *random search* as a baseline. We use the algorithm parameter values as suggested in the *DynaMOSA* paper[7]. We set a search budget of 180 seconds as often used in related work [10, 11]. To account for the stochastic nature of search-based approaches, each file under test was run 20 times. This resulted in 8.25 d of consecutive running time (3960 runs × 180 s). The experiment was performed on a system with two AMD EPYC 7H12 (64 cores, 2.6 GHz) CPUs and 512 GB of RAM.

To determine if one approach performs better than the others, we applied the unpaired Wilcoxon signed-rank test [3] with a threshold of 0.05. This non-parametric statistical test determines if two data distributions are significantly different. In addition, we apply the Vargha-Delaney $\hat{A}_{12}$ statistic [13] to determine the effect size of the result, which determines the magnitude of the difference between the two data distributions.

The results of our evaluation can also be found in Table 1. It shows the average branch coverage per benchmark project achieved by *random search* and *DynaMOSA* and how they perform compared to each other. As can be seen in the table, *DynaMOSA* achieves an average branch coverage above 70 % for four out of six projects, and close to 50 % for the remaining two. As shown in related work, *DynaMOSA* achieves higher code coverage than *random search* for most `units` under test. Additionally, Table 1 shows the statistical results of the comparison between the two search algorithms with regard to branch coverage across the various benchmarks. This section of the table is organized into three main categories: #Win, #No Diff, and #Lose. Analyzing the #Win category, we observe notable results in favor of *DynaMOSA* in all benchmarks. The table shows that in 41 cases *DynaMOSA* wins significantly, in 6 cases *random search* wins significantly, and in 52 cases there is no significant difference in performance.

4 CONCLUSION AND FUTURE WORK

In this paper, we introduced *SynTest-JavaScript*, a unit-level automated test case generation tool for (server-side) *JavaScript*. With this tool, we provide a platform for researchers to experiment with new search-based approaches for the dynamic programming language *JavaScript*. Additionally, as no tool existed for (server-side) *JavaScript*, we provide practitioners with a new tool to apply search-based testing techniques in industry.

As part of our future plan, we will extend the tool with additional search algorithms (*e.g.,* SPEA2, PESA, PSO) and LLM-based approaches. To make it easier for researchers to evaluate new approaches, we plan to provide infrastructure within the tool needed to easily run and compare experiments. Furthermore, we plan to incorporate a mutation-testing engine to better evaluate the quality of the test cases. Lastly, to make the tool easier to use for practitioners, we plan to integrate it within the most popular IDEs (*e.g.,* VSCode and WebStorm) and CI/CD platforms (*e.g.,* GitHub, GitLab).

[7] https://github.com/syntest-framework/syntest-framework/blob/3f6b9612c030ffc79d5e79c5c1c126ca816a87a6/tools/base-language/lib/presets/DynaMOSAPreset.ts

REFERENCES

[1] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of JavaScript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*. 571–580.

[2] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.

[3] William Jay Conover. 1998. *Practical nonparametric statistics*. Vol. 350. John Wiley & Sons.

[4] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[5] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: automatic symbolic testing of JavaScript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 449–459.

[6] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. *arXiv preprint arXiv:2202.05218* (2022).

[7] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[8] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013. PYTHIA: Generating test cases with oracles for JavaScript applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 610–615.

[9] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. JSeft: Automated JavaScript unit test generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10.

[10] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. 1–10.

[11] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[12] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2022. Guess What: Test Case Generation for Javascript with Unsupervised Probabilistic Type Inference. In *International Symposium on Search Based Software Engineering*. Springer, 67–82.

[13] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.