





Guess What: Test Case Generation for Javascript with Unsupervised Probabilistic Type Inference

Dimitri Stallenberg, Mitchell Olsthoorn^(✉) , and Annibale Panichella 

Delft University of Technology, Delft, The Netherlands

D.M.Stallenberg@student.tudelft.nl,

{M.J.G.Olsthoorn,A.Panichella}@tudelft.nl

Abstract. Search-based test case generation approaches make use of static type information to determine which data types should be used for the creation of new test cases. Dynamically typed languages like JavaScript, however, do not have this type information. In this paper, we propose an unsupervised probabilistic type inference approach to infer data types within the test case generation process. We evaluated the proposed approach on a benchmark of 98 units under test (*i.e.*, exported classes and functions) compared to random type sampling *w.r.t.* branch coverage. Our results show that our type inference approach achieves a statistically significant increase in 56% of the test files with up to 71% of branch coverage compared to the baseline.

Keywords: Empirical software engineering · Search-based software testing · Test case generation · Javascript · Type inference

1 Introduction

Over the last few decades, researchers have developed various techniques for automating test case generation [31]. In particular, search-based approaches have been shown to (1) achieve higher code coverage [25] and (2) have fewer smells [37] compared to manually-written test cases, and (3) detect unknown bugs [1, 2, 21]. Furthermore, generated tests significantly reduce the time needed for testing and debugging [42], and have been successfully used in industry (*e.g.*, [3, 11, 30]).

These approaches make use of static type information to (1) generate primitives and objects to pass to constructors and function calls, and (2) determine which branch distance function to use. Without this type information, the test case generation process has to randomly guess which types are compatible with the parameter specification of the constructor or function call and would not have guidance to solve the binary flag problem. This greatly increases the search space and, therefore, makes the overall process less effective and efficient. Consequently, most of the work in this research area has focused on statically-typed programming languages like Java (*e.g.*, EVOSUITE [18]) and C (*e.g.*, AUSTIN [26]).

Dynamically-typed programming languages introduce new challenges for unit-level test case generation. As reported by Lukasczyk *et al.* [28], state-of-the-art approaches used for statically-typed languages do not perform well on Python programs when type information is not available. According to the survey from Stack Overflow¹, Python and JavaScript are the two most commonly-used programming languages. Both languages are dynamically-typed, strengthening the importance of addressing these open challenges with the goal of increasing the adoption of test case generation tools in general.

In this paper, we focus on test case generation for JavaScript as, to the best of our knowledge, this is a research gap in the literature. In building our research, we build on top of the reported experience by Lukasczyk *et al.* [29] for Python programs. They addressed the input type challenge by incorporating Type4Py [32]—a deep neural network (DNN)—into the search process.

We propose a novel approach that incorporates unsupervised probabilistic type inference into the search-based test case generation process to infer the type information needed. An unsupervised type inference approach has two benefits compared to a DNN: (1) it does not require a labeled dataset with extensive training time, and (2) the model is explainable (*i.e.*, the decision can be traced back to a rule set). We build a prototype tool which implements the state-of-the-art many-objective search algorithm, DYNAMOSA, and the probabilistic type inference model for JavaScript. We investigate two different strategies for incorporating the probabilistic model into the main loop of DYNAMOSA, namely *proportional sampling* and *ranking*.

To evaluate the performance of the proposed approach, we performed an empirical study that investigates the baseline performance of our prototype (*i.e.*, using random type sampling) and the impact of the unsupervised probabilistic type inference *w.r.t.* branch coverage. To this aim, we constructed a benchmark consisting of 98 Units under Test (*i.e.*, exported classes and functions) of five popular open-source JavaScript projects, namely **Commander.js**, **Express**, **Moment.js**, **Javascript Algorithms**, and **Lodash**.

Our results show that integrating unsupervised probabilistic type inference improves branch coverage compared to random type sampling. Both the *ranking* and *proportional sampling* strategies significantly increase the number of branches covered by our approach (+9.3% and +12.6%, respectively). Out of the two strategies, *proportional sampling* outperforms *ranking* in 20 cases and loses in 4. In summary, we make the following contributions:

1. An unsupervised probabilistic type inference approach for search-based unit-level test case generation of JavaScript programs.
2. A prototype tool for automatically generating JavaScript unit-level test cases that incorporates this approach.²
3. A Benchmark consisting of 98 units under test from five popular open-source JavaScript projects.
4. A full replication package containing the results and the analysis scripts [43].

¹ <https://survey.stackoverflow.co/2022/#most-popular-technologies-language>.

² <https://github.com/syntest-framework/syntest-javascript>.

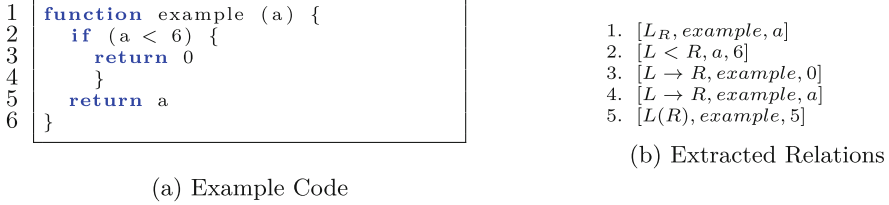
2 Background and Related Work

This section explains the background concepts and discusses the related work.

Test Case Generation. Writing test cases is an expensive, tedious, yet necessary activity for software quality assurance. Hence, researchers have proposed various techniques to semi-automate this process since the 1970s [15]. These techniques include symbolic execution [10], random testing [14], and meta-heuristics [31] (*e.g.*, genetic algorithms). The latter category is often referred to as search-based software testing (SBST). SBST techniques have been successfully used in the literature to automate the creation of test cases for different testing levels [31], such as unit [19], integration [17], and system-level testing [6]. At unit-level, SBST techniques aim to generate test cases that optimize various test adequacy criteria, such as *e.g.*, structural coverage and mutation score. Many different meta-heuristic search algorithms have been proposed over the years (*e.g.*, whole suite [20], MIO [5], MOSA [38], or DynaMOSA [39]). Recent studies have shown that DYNAMOSA is more effective and efficient than other genetic algorithms for unit test generation of Java [12] and Python [28] programs.

Type Inference. A recent study by Gao *et al.* [22] showed that the lack of static types within JavaScript leads to bugs that could have been easily identified with a static type system. To combat this problem, various approaches have been proposed to infer/predict types for generating type annotations or assertions. Anderson *et al.* [4] proposed a formal approach for inferring types using constraint solvers based on a custom JavaScript-like language. Chandra *et al.* [13] proposed a formal type inference approach for static compilation of JavaScript programs. These approaches, however, only support a subset of the JavaScript syntax and, therefore, will not work on all programs. JSNice [41] and DeepTyper [24] are two other approaches that train a model based on training data and use it to predict future type information. These approaches have the shortcoming that they can only predict basic JavaScript types. Meaning that they are unable to predict/assert user-defined types. Additionally, these approaches cannot consider the context of the literals and objects within a program or function. Type4Py [32] is a similar approach that uses a Deep Neural Network (DNN) to infer types for Python projects and suffers from similar limitations.

Testing for JavaScript. JavaScript started out as a client-side programming language for the browser. Most work related to testing for JavaScript is, therefore, also focused on web applications within the browser (*e.g.*, [9, 27, 34, 44]). Existing client-side testing approaches either focus on specific subsystems such as the browser's event handling system [9, 27] or the interaction of JavaScript with the *Document Object Model* of the browser [34]. Nowadays, JavaScript is also a very commonly-used language for back-end development on *Node.js*. Tanida *et al.* [44] proposed a symbolic execution approach that uses a constraint solver for input data generation. Other approaches focused on mutation testing [33] or contract-based testing [23]. However, to the best of our knowledge, there exists no approach for automatic unit-level test case generation for JavaScript.

**Fig. 1.** Extracting relations from code

3 Approach

This section details our test case generation approach for JavaScript programs that relies on Unsupervised Type Inference. Our approach consists of three phases, which are detailed in the next subsections.

3.1 Phase 1: Static Analysis

The first phase inspects the Subject Under Test (SUT) and its dependencies. First, this phase builds the Abstract Syntax Trees (ASTs) and extracts all identifiers and literals from the code; these will be referred to as *elements*. Afterward, the static analyzer extracts the relations between those elements and all user-defined objects, i.e., classes, interfaces, or prototyped objects.

Elements. As mentioned before, the elements consist of *identifiers* and *literals*. The former are the named references to variables, functions, and properties. The latter are constant values assigned to variables; examples are strings, numbers, and booleans. The types of the literal are straightforward and do not require inference. However, the identifiers do not have explicit types in dynamically typed languages like JavaScript. Hence, their types need to be inferred based on the *contextual* information (or *relations*) of the extracted elements.

Relations. Relations correspond to operations performed on code elements and describe how these elements are used and relate to other elements, providing hints on their types. For example, let us consider the *assignment* relation $L = R$, where R (right-hand element) is a boolean literal; we can logically derive (or infer) that L (left-hand element) must also be a boolean variable.

These relations are extracted from the AST and are converted to a consistent format that allows for easy identification of the relation type. Let us assume that there is a *lower than* relation between variable a and literal 6 , as shown in Fig. 1a on line 2. This relation is converted and recorded as $[L < R, a, 6]$, as shown in Fig. 1b. In general, a relation is stored as a tuple containing (1) the type of operation ($L < R$ in our example) and (2) the list of operands (i.e., a and 6 in our example). The full list of extracted relations for the code snippet in Fig. 1a is reported in Fig. 1b.

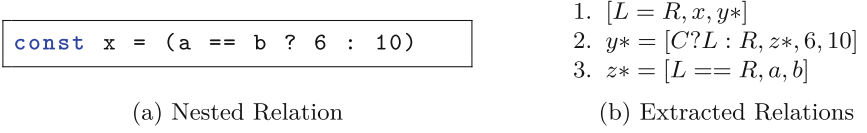


Fig. 2. Extracting relations from nested code

In total, we designed 75 possible relations based on the MDN web documentation by Mozilla³. These operations/relations are classified into 15 categories, namely (1) *primary*, (2) *left-hand side*, (3) *increment/decrement*, (4) *unary*, (5) *arithmetic*, (6) *relational*, (7) *equality*, (8) *bitwise shift*, (9) *binary bitwise*, (10) *binary logical*, (11) *ternary*, (12) *optional chaining*, (13) *assignment*, (14) *comma*, and (15) *function* expressions. The complete list of relations is available in our replication package.

Nested relations are special types of relations whose composing elements are relations themselves. As an example, let us consider the code snippet in Fig. 2a. The corresponding relation for the assignment is $[L = R, x, y^*]$, where y^* is an *artificial* element that points to the whole right-hand side of the assignment. This element corresponds to a ternary relation $[C?L : R, z^*, 6, 10]$, which also includes an artificial element, called z^* , that points to the equality relation in the conditional part of the ternary statement. So z^* points to the final relation $[L == R, a, b]$. Although the code in Fig. 2a seems rather simple, it corresponds to three relations, two of which are nested, as shown in Fig. 2b.

Scopes. A critical aspect of the elements we have not yet discussed is scoping. The scope of an identifier determines its accessibility. To better understand the importance of the scope, let us consider the example in Fig. 3. First, the constant x is assigned the value 5. The constant x is defined in the so-called *global scope*. Next, a function is defined, creating a new scope. This scope has access to references of the global scope. Still, it can also have its own references, which are only available within its sub-scopes. In our example, another constant x is defined within the function scope. Note that from line 4, every reference to x in the scope of the function refers to the newly defined constant, not the x constant of the global scope. This type of operation is called *variable shadowing*. In a nutshell, variable shadowing is when the code contains an identifier for which there are multiple declarations in separate scopes. In these situations, the narrower scope *shadows* the other identifier declarations.

This shadowing principle is fundamental during the first phase of our approach because variables in the global scope are not the same variables as those in the function scope (e.g., x in Fig. 3). In fact, variables with the same identifier names but within different scopes can also have different types. In the example of Fig. 3, x from the global scope is numerical, while the x from the function scope is a string. In conclusion, the relations include the involved elements together with their scope.

³ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators>.

```

1  const x = 5
2
3  function example(a) {
4      const x = "Hello "
5      return x + a
6  }

```

Fig. 3. Scopes

Complex Objects. In JavaScript, objects are the building blocks of the language and are stored as key-value pairs. Apart from primitive types like booleans and numbers, almost everything is represented as an object. An array, for example, is a special object where the keys are numbers. In recent JavaScript versions, developers can define classes and interfaces through a prototype-based object model, inducing a more object-oriented approach to JavaScript. Since these objects play such a prominent role in JavaScript, it is important that object types can be inferred as well. Hence, our approach extracts all object descriptions available in the program under test, including class, interface definitions, and standard objects (e.g., functions).

3.2 Phase 2: Unsupervised Static Type Inference

The second phase builds a probabilistic type model for the elements extracted from the first phase. For literal elements, the type inference is straightforward as the type can be directly inferred from the literal type. However, for non-literal elements, our probabilistic model considers all *type hints* that can be inferred from the relations extracted in the previous phase.

For example, the assignment `x = 5` corresponds to the relation $[L = R, x, 5]$. From such a relation, we can derive that, at this particular point in the code, `x` must be numerical since it is assigned the literal value 5. However, for statements like `x = y + z`, there are various possibilities for the type of `x` depending on the types of `y` and `z`. To illustrate, the `+` operator can be applied to both numbers (arithmetic sum) and strings (string concatenation). Besides, in JavaScript, it is also possible to concatenate numbers with strings. For example, `1 + "1"` returns the number 11. Therefore, multiple types can be assigned to elements that have relations/operations compatible with multiple data types.

To account for this, our model assigns scores to each type depending on the number of hints that can be derived for that type by its relations in the code. In general, given the element e and the set of relations $R = \{r_1, \dots, r_n\}$ associated to e as extracted from a program P , our model assigns each type t a score equal to the number of relations that can be applied to t (i.e., the number of hints):

$$\text{score}(e, t) = |\text{hints}(e, t)| \quad \text{where} \quad \text{hints}(e, t) = \{r_i \in R : r_i \text{ applies to } t\} \quad (1)$$

Finally, the element e has a probability of being assigned the type t proportional to the number of hints received for t :

$$p(e, t) = \frac{\text{score}(e, t)}{\sum_{t_i} \text{score}(e, t_i)} \quad (2)$$

The higher the score of a particular type, the larger the probability that the element is of that type. The probabilities are later used to sample argument types in the search phase.

For example, let us consider the statement $x = y + z$, which can be applied to both strings and numbers. In this case, our probabilistic model would assign +1 hint for numbers and +1 hint for strings. Hence, both types will have an equal probability of 50%.

Nested Types. The probability model takes into account both simple and nested relations. For example, let us consider the JavaScript statement: $c = a > b$. Such a statement corresponds to two relations (one of which is nested): $[L = R, c, d^*]$ and $d^* = [L > R, a, b]$. The outcome for $d^* = [L > R, a, b]$ is boolean no matter the types of a and b . Therefore, we can infer the variable (or element) c should be as well. Hence, the hints and scores are obtained by considering all relations, including the nested ones.

Resolving Complex Objects. Complex objects are characterized by *property accessor* relations, i.e., operations that aim to access properties of certain objects (e.g., using the dot notation `object.property`). If an element is involved in one or more *property accessor* relations, the accessed properties are compared to the available object descriptions. If there is an overlap between the element's properties and the properties of an object description, the object description receives +1 hint. In addition to matching object descriptions, an anonymous object type is created and assigned as a possible type. This anonymous object type exactly matches the properties of the element. This object is used when no other object matches are found.

3.3 Phase 3: Test Case Generation

The third phase generates test cases using meta-heuristics with the goal of maximizing branch coverage. As explained in Sect. 2, we use the Dynamic Many-Objective Sorting Algorithm (DynaMOSA) [39] as suggested in the literature [12, 29, 40]. Previous studies have shown that DYNAMOSA outperforms other meta-heuristics in unit test case generation for Java [12, 40], python [29], and solidity [36] programs. Assessing other meta-heuristics in the context of unit test generations for JavaScript programs is part of our future agenda.

Our implementation applies the probabilistic model described in Sect. 3.2 to determine what is the potential type of each input parameter. We have implemented two different strategies to incorporate the type inference model into the main DYNAMOSA loop, namely *proportional type sampling* and *ranking*.

Table 1. Benchmark statistics

Benchmark	#Units	CC	SLOC	Avg. n. branches
Commander.js	4	23	208	29
Express	15	20	222	25
Moment.js	54	7	33	8
Javascript Algorithms	30	5	68	8
Lodash	10	11	63	16

Proportional Sampling. This strategy can assign various types to each input parameter. As explained in Sect. 3.2, our model assigns scores to multiple types (see Eq. (1)) based on the number of positive hints received by analyzing the associated relations. When creating a new test case (either in the initial population or during mutation), each input parameter is assigned one of the types. Each candidate type has a probability of being selected equal to the value obtained by applying Eq. (2). Notice that each data type is sampled for each newly generated test case. Therefore, the same input parameter (for the same function) may be assigned different types every time a new test case is created.

Ranking. This strategy assigns only one type to the input parameter. In particular, this strategy sorts all types with positive hints in descending order of their score values. Then, this method selects the type with the largest probability (or the largest number of hints).

Test Execution. Once generated, each generated test case will contain a sequence of function calls with their input data. These tests are then executed against the program under test, and the coverage information is stored. The “fitness” of a test is measured according to its distance to cover all unreached branches in the code, as typically done in DYNAMOSA. The distance to each uncovered branch is computed using two well-known coverage heuristics [31]: (1) the *approach level* and (2) the normalized *branch distance*.

4 Empirical Study

To assess the impact of the unsupervised probabilistic type inference on the performance of search-based unit test generation for JavaScript, we perform an empirical evaluation to answer the following research questions:

RQ1 *How does unsupervised static type inference impact structural coverage of DYNAMOSA for JavaScript?*

RQ2 *What is the best strategy to incorporate type inference in DYNAMOSA?*

Benchmark. To the best of our knowledge, there is no existing JavaScript benchmark for unit-level test case generation. Hence, for our empirical study,

we build a benchmark comprising of five JavaScript projects: *Express*⁴, *Commander.js*⁵, *Moment.js*⁶, *JavaScript Algorithms*⁷, *Lodash*⁸. These projects were selected based on their popularity in the JavaScript community (measured through the number of stars on GitHub) and represent a diverse collection of JavaScript syntax and code styles. From these projects, we selected a subset of **units** (*i.e.*, classes or functions) based on two criteria: (1) the unit has to be testable (*i.e.*, the unit has to be exported), and (2) the unit needs to be non-trivial (*i.e.*, have a Cyclomatic Complexity of $CC \geq 2$ as calculated by *Plato*⁹). The latter criterion is in line with existing guidelines for assessing test case generation tools [40]. Table 1 provides the main characteristics of our benchmark at the project-level, including the average Cyclomatic Complexity per project (**CC** column), the average Source Lines Of Code (**SLOC** column), and the average number of branches. It is worth noting that some of the files in the selected projects had to be excluded or modified. For example, in the **Commander.js** project there are two files that contain statements that terminate the running process. This has the effect of also terminating the test case generation process. Therefore, we have excluded this file from the benchmark and modified it, so that any other files depending on it will not be affected.

Prototype. To evaluate the proposed approach, we have developed a prototype for unit-level test case generation that implements our unsupervised dynamic type inference, written in **Typescript**. The prototype also implements the state-of-the-art search algorithm for test case generation, namely DYNAMOSA [39], as well as the guiding heuristics [31], *i.e.*, the approach level and branch distance.

Parameter Settings. For this study, we have chosen to mainly adopt the default search algorithm parameter values as described in literature [39]. Previous studies have shown that although parameter tuning impacts the search algorithm’s performance, the default parameter values provide reasonable and acceptable results [8]. Hence, the search algorithm uses a single point crossover with a crossover probability of 0.75, mutation with a probability of $1/n$ (n = number of statements in the test case), and tournament selection. For the population size, however, we decided to deviate from the default (50). We went for a size of 30 as our preliminary experiment showed this worked best for a benchmark this size. The search budget per unit under test is 60 s. This is a common value used in related work [35].

Experimental Protocol. To answer RQ1, we compare the two variants of our approach with DYNAMOSA without type inference. In particular, for this baseline, the type for the input data is randomly sampled among all types that can be extracted using the relations described in Sect. 3.1. To answer RQ2, we

⁴ <https://expressjs.com/>.

⁵ <https://tj.github.io/commander.js/>.

⁶ <https://momentjs.com/>.

⁷ <https://github.com/trekhleb/javascript-algorithms>.

⁸ <https://lodash.com/>.

⁹ <https://github.com/es-analysis/plato>.

compare the two variants of our approach: (1) proportional type sampling, and (2) ranking.

To account for the stochastic nature of the approach, each unit under test was run 20 times. We performed 20 repetitions of 3 configurations (*i.e.*, random type sampling, ranking, and proportional sampling) on 98 units under test, for a total of 5880 runs. This required $(5880 \text{ runs} \times 60 \text{ s}) / (60 \text{ s} \times 60 \text{ min} \times 24 \text{ h}) \approx 4.1 \text{ d}$ computation time. At the end of each run, we stored the maximum branch coverage achieved by the approach for the active configuration (**RQ1** and **RQ2**). The experiment was performed on a system with an AMD Ryzen 9 3900X (12 cores 3.8 GHz) with 32 GB of RAM. Each experiment was given a maximum of 8 GB of RAM. To determine if one approach performs better than the others, we applied the unpaired Wilcoxon signed-rank test [16] 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 [45] to determine the effect size of the result, which determines the magnitude of the difference between the two data distributions.

5 Results

This section discusses the results of our empirical study with the aim of answering the research questions formulated in Sect. 4. All differences in results are presented in absolute differences (percentage points).

Result for RQ1: Structural Coverage. Table 2 summarizes the results achieved by our approach on the benchmark with the winning configuration highlighted in gray color. It shows the median branch coverage and the Inter-Quartile-Range (IQR) for the two possible strategies to incorporate the type inference model (Ranking, Proportional) and a baseline that uses random type sampling (Random). The *Units* column indicates the number of units (*i.e.*, exported classes and functions) that are tested in the file of the benchmark project.

On average for all 57 files in the benchmark, *random* achieves 33.4% branch coverage, *ranking* 42.7%, and *proportional type sampling* 46.0%. The baseline still performs quite well, as *random type sampling* can be effective in triggering assertion branches and can over time guess the correct types for primitives. For the *ranking* strategy, the average improvement in branch coverage is 9.3%. The file with the least improvement is `suggestSimilar.js` from the `Commander.js` project with an average decrease of 13%. The file with the most improvement is `add-subtract.js` from the `Moment.js` project with an average increase of 71%, which corresponds to 10 additionally covered branches. For the *proportional* strategy, the average improvement in branch coverage is 12.6%. There are 24 files for which the *proportional* strategy performs equally to the baseline. The file with the most improvement is again `add-subtract.js` from the `Moment.js` project with an average increase of 71%.

Table 3 shows the results of the statistical comparison between the two strategies and the baseline, based on a p -value ≤ 0.05 . *#Win* indicates the number of

Table 2. Median branch coverage and the inter-quartile-range. The largest values are highlighted in gray color.

Benchmark	File Name	#Units	Random		Ranking		Proportional	
			Median	IQR	Median	IQR	Median	IQR
Commander.js	help.js	1	0.20	0.0190	0.41	0.0760	0.53	0.023
	option.js	2	0.33	0.0560	0.33	0.0560	0.39	0.000
	suggestSimilar.js	1	0.69	0.0620	0.56	0.1560	0.75	0.062
Express	application.js	1	0.63	0.0190	0.63	0.0190	0.65	0.019
	query.js	1	0.67	0.0000	0.67	0.0000	0.67	0.000
	request.js	1	0.25	0.0000	0.27	0.0230	0.25	0.023
	response.js	1	0.14	0.0070	0.13	0.0130	0.14	0.013
	utils.js	7	0.56	0.0070	0.62	0.0000	0.59	0.029
	view.js	1	0.06	0.0000	0.06	0.0000	0.06	0.000
JS Algorithms Graph	articulationPoints.js	1	0.00	0.0000	0.00	0.0000	0.08	0.000
	bellmanFord.js	1	0.00	0.0000	0.17	0.0000	0.33	0.000
	bfTravellingSalesman.js	1	0.00	0.0000	0.08	0.0000	0.08	0.000
	breadthFirstSearch.js	1	0.12	0.1250	0.38	0.0310	0.31	0.125
	depthFirstSearch.js	1	0.00	0.1670	0.00	0.1670	0.00	0.167
	detectDirectedCycle.js	1	0.00	0.0000	0.12	0.0000	0.38	0.000
	dijkstra.js	1	0.00	0.0000	0.10	0.0000	0.20	0.100
	eulerianPath.js	1	0.00	0.0000	0.00	0.0000	0.21	0.000
	floydWarshall.js	1	0.00	0.0000	0.67	0.0000	0.67	0.000
	hamiltonianCycle.js	1	0.00	0.0000	0.00	0.0000	0.00	0.050
	kruskal.js	1	0.10	0.1000	0.30	0.0000	0.30	0.000
	prim.js	1	0.08	0.0000	0.08	0.0830	0.17	0.000
	stronglyConnectedComponents.js	1	0.00	0.0000	0.00	0.0000	0.25	0.000
JS Algorithms Knapsack	Knapsack.js	1	0.57	0.0000	0.50	0.0000	0.57	0.000
	KnapsackItem.js	1	0.50	0.0000	0.50	0.0000	0.50	0.000
JS Algorithms Matrix	Matrix.js	12	0.79	0.0530	0.74	0.0260	0.80	0.158
JS Algorithms Sort	CountingSort.js	1	0.92	0.0830	0.92	0.0210	0.92	0.000
JS Algorithms Tree	RedBlackTree.js	1	0.21	0.0000	0.26	0.0000	0.29	0.037
Lodash	equalArrays.js	1	0.08	0.0000	0.67	0.0420	0.75	0.052
	hasPath.js	1	0.75	0.1560	0.75	0.0000	0.88	0.250
	random.js	1	1.00	0.0001	1.00	0.0001	1.00	0.000
	result.js	1	0.90	0.1000	0.80	0.0000	0.90	0.100
	slice.js	1	1.00	0.0001	1.00	0.0001	1.00	0.000
	split.js	1	0.88	0.0000	0.88	0.0000	0.88	0.000
	toNumber.js	1	0.60	0.0000	0.65	0.0000	0.65	0.050
	transform.js	1	0.83	0.0000	0.83	0.0000	0.83	0.083
	truncate.js	1	0.38	0.0000	0.59	0.0290	0.59	0.000
	unzip.js	1	1.00	0.0001	1.00	0.0001	1.00	0.000
Moment.js	add-subtract.js	1	0.00	0.0000	0.71	0.0180	0.71	0.000
	calendar.js	2	0.05	0.0000	0.45	0.0910	0.43	0.091
	check-overflow.js	1	0.05	0.0000	0.60	0.0000	0.60	0.000
	compare.js	6	0.14	0.0000	0.14	0.0000	0.14	0.000
	constructor.js	3	0.38	0.0000	0.53	0.0080	0.41	0.156
	date-from-array.js	2	0.88	0.0000	0.88	0.0000	0.88	0.000
	format.js	4	0.08	0.0000	0.08	0.0000	0.08	0.000
	from-anything.js	2	0.68	0.0590	0.71	0.0000	0.69	0.037
	from-array.js	1	0.02	0.0000	0.04	0.0000	0.04	0.000
	from-object.js	1	0.50	0.0000	0.50	0.0000	0.50	0.000
	from-string-and-array.js	1	0.00	0.0000	0.31	0.0000	0.31	0.000
	from-string-and-format.js	1	0.06	0.0000	0.56	0.0390	0.55	0.133
	from-string.js	3	0.06	0.0000	0.16	0.0000	0.16	0.000
	get-set.js	5	0.14	0.0000	0.23	0.0450	0.36	0.068
	locale.js	2	0.17	0.1670	0.17	0.0000	0.17	0.000
	min-max.js	2	0.12	0.0000	0.12	0.0000	0.12	0.000
	now.js	1	0.50	0.0000	0.50	0.0000	0.50	0.000
	parsing-flags.js	1	0.50	0.0000	0.50	0.1250	0.50	0.000
	start-end-of.js	2	0.10	0.0000	0.10	0.0000	0.10	0.000
	valid.js	2	0.38	0.0000	0.38	0.0000	0.38	0.000

Table 3. Statistical results w.r.t. branch coverage

Comparison	#Win				#No diff.		#Lose			
	Negl.	Small	Medium	Large	Negl.	Negl.	Small	Medium	Large	
Ranking vs. Random	-	3	1	23	26	-	1	-	3	
Prop. sampling vs. Random	-	1	4	27	25	-	-	-	-	
Prop. sampling vs. Ranking	-	4	-	16	33	-	3	1	-	

times that the left configuration has a statistically significant improvement over the right one. *#No diff.* indicates the number of times that there is no evidence that the two competing configurations are different; *#Lose* indicates the number of times that the left configuration has statistically worse results than the right one. The *#Win* and *#Lose* columns also include the \hat{A}_{12} effect size, classified into *Small*, *Medium*, *Large*, and *Negligible*.

We can see that the *ranking* and the *proportional* strategy have a statistically significant non-negligible improvement over the baseline in 27 and 32 files for branch coverage, respectively. *Ranking* improves with a large magnitude for 23 classes, medium for 1 class, and small for 3 classes and *proportional* with 27 (large), 4 (medium), and 1 (small). The *Ranking* strategy loses in four cases when compared to the baseline: `response.js`, `response.js`, `Knapsack.js`, `Matrix.js`, and `results.js`.

Result for RQ2: Strategy. When we compare the two different strategies with each other, we can observe that the *proportional type inference* on average improves by 3.3% over the *ranked* strategy based on branch coverage. The file with the least improvement is `constructor.js` from the `Moment.js` project with an average decrease of 12%. While the file with the most improvement is `detectDirectedCycle.js` from the `JS Algorithms` project with an average increase of 36%. From Table 3, we can see that the *proportional* strategy has a statistically significant non-negligible improvement over *ranking* in 20 cases (16 large and 4 small). While *ranking* improves over *proportional* in only 4 cases (1 medium and 3 small): `slice.js`, `constructor.js`, `from-string-and-format.js`, and `parsing-flags.js`.

6 Threats to Validity

This section discusses the potential threats to the validity of our study.

External Validity: An important threat regards the generalizability of our study. We selected five open-source projects based on their popularity in the JavaScript community. The projects are diverse in terms of size, application domain, purpose, syntax, and code style. Further experiments on a larger set of projects would increase the confidence in the generalizability of our study and, therefore, is part of our future work.

Conclusion Validity: Threats to *conclusion validity* are related to the randomized nature of DYNAMOSA. To minimize this risk, we have executed each

configuration 20 times with different random seeds. We have followed the best practices for running experiments with randomized algorithms as laid out in well-established guidelines [7]. Additionally, we used the unpaired Wilcoxon signed-rank test and the Vargha-Delaney \hat{A}_{12} effect size to assess the significance and magnitude of our results. To ensure a controlled environment that provides a fair evaluation, all experiments have been conducted on the same system and interfering processes were kept to a minimum.

7 Conclusion and Future Work

In this paper, we presented an automated unit test generation approach for JavaScript, the most popular dynamically-typed language. It generates unit-level test cases by using the state-of-the-art meta-heuristic search algorithm DYNAMOSA and a novel unsupervised probabilistic type inference model. Our results show that (1) the proposed approach can successfully generate test cases for well-established libraries in JavaScript, and (2) the type inference model plays a significant role in achieving larger code coverage (through *proportional sampling*). As part of our future work, we plan (1) to extend our benchmark, (2) to investigate more meta-heuristics, (3) assess different strategies to incorporate the type inference model within the search process, and (4) compare our type inference model to state-of-the-art deep learning approaches.

References

1. Abdesslem, R.B., Panichella, A., Nejati, S., Briand, L.C., Stifter, T.: Testing autonomous cars for feature interaction failures using many-objective search. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 143–154 (2018)
2. Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J.: An industrial evaluation of unit test generation: finding real faults in a financial application. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 263–272 (2017)
3. Alshahwan, N., et al.: Deploying search based software engineering with Sapienz at Facebook. In: Colanzi, T.E., McMin, P. (eds.) SSBSE 2018. LNCS, vol. 11036, pp. 3–45. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99241-9_1
4. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 428–452. Springer, Heidelberg (2005). https://doi.org/10.1007/11531142_19
5. Arcuri, A.: Test suite generation with the many independent objective (MIO) algorithm. *Inf. Softw. Technol.* **104**, 195–206 (2018)
6. Arcuri, A.: RESTful API automated test case generation with EvoMaster. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **28**(1), 1–37 (2019)
7. Arcuri, A., Briand, L.: A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verif. Reliab.* **24**(3), 219–250 (2014)

8. Arcuri, A., Fraser, G.: Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empir. Softw. Eng.* **18**(3), 594–623 (2013)
9. Artzi, S., Dolby, J., Jensen, S.H., Møller, A., Tip, F.: A framework for automated testing of JavaScript web applications. In: *Proceedings of the 33rd International Conference on Software Engineering*, pp. 571–580 (2011)
10. Baldoni, R., Coppa, E., D’elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. *ACM Comput. Surv. (CSUR)* **51**(3), 1–39 (2018)
11. Ben Abdesslem, R., Nejati, S., Briand, L.C., Stifter, T.: Testing advanced driver assistance systems using multi-objective search and neural networks. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 63–74 (2016)
12. Campos, J., Ge, Y., Albulian, N., Fraser, G., Eler, M., Arcuri, A.: An empirical evaluation of evolutionary algorithms for unit test suite generation. *Inf. Softw. Technol.* **104**, 207–235 (2018)
13. Chandra, S., et al.: Type inference for static compilation of JavaScript. *ACM SIGPLAN Not.* **51**(10), 410–429 (2016)
14. Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In: Maher, M.J. (ed.) *ASIAN 2004. LNCS*, vol. 3321, pp. 320–329. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30502-6_23
15. Clarke, L.A.: A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.* **3**, 215–222 (1976)
16. Conover, W.J.: *Practical Nonparametric Statistics*, vol. 350. Wiley, Hoboken (1998)
17. Derakhshanfar, P., Devroey, X., Panichella, A., Zaidman, A., van Deursen, A.: Towards integration-level test case generation using call site information. *arXiv preprint arXiv:2001.04221* (2020)
18. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 416–419 (2011)
19. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011*, pp. 416–419. ACM, New York (2011). <https://doi.org/10.1145/2025113.2025179>
20. Fraser, G., Arcuri, A.: Whole test suite generation. *IEEE Trans. Softw. Eng.* **39**(2), 276–291 (2012)
21. Fraser, G., Arcuri, A.: 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empir. Softw. Eng.* **20**(3), 611–639 (2015)
22. Gao, Z., Bird, C., Barr, E.T.: To type or not to type: quantifying detectable bugs in JavaScript. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 758–769 (2017)
23. Heidegger, P., Thiemann, P.: Contract-driven testing of JavaScript code. In: Vitek, J. (ed.) *TOOLS 2010. LNCS*, vol. 6141, pp. 154–172. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13953-6_9
24. Hellendoorn, V.J., Bird, C., Barr, E.T., Allamanis, M.: Deep learning type inference. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 152–162 (2018)
25. Kifetew, F., Devroey, X., Rueda, U.: Java unit testing tool competition-seventh round. In: *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pp. 15–20 (2019)

26. Lakhotia, K., Harman, M., Gross, H.: AUSTIN: an open source tool for search based software testing of c programs. *Inf. Softw. Technol.* **55**(1), 112–125 (2013)
27. Li, G., Andreasen, E., Ghosh, I.: SymJS: automatic symbolic testing of JavaScript web applications. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 449–459 (2014)
28. Lukasczyk, S., Kroiß, F., Fraser, G.: Automated unit test generation for Python. In: Aleti, A., Panichella, A. (eds.) *SSBSE 2020. LNCS*, vol. 12420, pp. 9–24. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59762-7_2
29. Lukasczyk, S., Kroiß, F., Fraser, G.: An empirical study of automated unit test generation for python. *arXiv preprint arXiv:2111.05003* (2021)
30. Matinnejad, R., Nejati, S., Briand, L.C., Bruckmann, T.: Automated test suite generation for time-continuous simulink models. In: *proceedings of the 38th International Conference on Software Engineering*, pp. 595–606 (2016)
31. McMin, P.: Search-based software test data generation: a survey. *Softw. Test. Verif. Reliab.* **14**(2), 105–156 (2004)
32. Mir, A.M., Latoškinas, E., Proksch, S., Gousios, G.: Type4Py: practical deep similarity learning-based type inference for Python. In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 2241–2252 (2022)
33. Mirshokraie, S., Mesbah, A., Pattabiraman, K.: Efficient JavaScript mutation testing. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 74–83 (2013)
34. Mirshokraie, S., Mesbah, A., Pattabiraman, K.: JSeft: automated JavaScript unit test generation. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10 (2015)
35. Olsthoorn, M., van Deursen, A., Panichella, A.: Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1224–1228 (2020)
36. Olsthoorn, M., Stallenberg, D., van Deursen, A., Panichella, A.: SynTest-solidity: automated test case generation and fuzzing for smart contracts. In: *The 44th International Conference on Software Engineering-Demonstration Track* (2022)
37. Panichella, A., Panichella, S., Fraser, G., Sawant, A.A., Hellendoorn, V.: Test smells 20 years later: detectability, validity, and reliability. *Empir. Softw. Eng.* **27**(7) (2022). <https://doi.org/10.1007/s10664-022-10207-5>
38. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10 (2015)
39. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Trans. Softw. Eng.* **44**(2), 122–158 (2017)
40. Panichella, A., Kifetew, F.M., Tonella, P.: A large scale empirical comparison of state-of-the-art search-based test case generators. *Inf. Softw. Technol.* **104**, 236–256 (2018)
41. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from “big code”. *ACM SIGPLAN Not.* **50**(1), 111–124 (2015)
42. Soltani, M., Panichella, A., Van Deursen, A.: Search-based crash reproduction and its impact on debugging. *IEEE Trans. Softw. Eng.* **46**(12), 1294–1317 (2018)
43. Stallenberg, D., Olsthoorn, M., Panichella, A.: Replication package of “guess what: test case generation for Javascript with unsupervised probabilistic type inference” (2022). <https://doi.org/10.5281/zenodo.7088684>

44. Tanida, H., Uehara, T., Li, G., Ghosh, I.: Automated unit testing of JavaScript code through symbolic executor SymJS. *Int. J. Adv. Softw.* **8**(1), 146–155 (2015)
45. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *J. Educ. Behav. Stati.* **25**(2), 101–132 (2000)