Infra-estrutura de Testes para Implementações de Referência do Standard ECMAScript

Diogo Costa Reis
ist187526
diogo.costa.reis@tecnico.ulisboa.pt

Instituto Superior Técnico
Av. Rovisco Pais, 1
1049-001 Lisboa
Tel: +351 218 417 000
mail@tecnico.ulisboa.pt

Abstract. **Keywords:** ECMAScript · Specification Language · Reference Interpreters · Test262

Table of Contents

1 Introduction . . . 3
2 Goals . . . 3
3 Background . . . 3
  3.1 ECMAScript . . . 3
  3.2 Test262 . . . 6
  3.3 An Infrastructure for testing reference implementations of the ECMAScript standard . . . 9
4 Related Work . . . 13
5 Design and Methodology . . . 13
6 Evaluation and Planning . . . 13
7 Conclusion . . . 13

1 Introduction

2 Goals

3 Background

This chapter provides an overview on the ECMAScript standard, the Test262 that are used to test the correct implementation of the ECMAScript standard, and finally an outline of the new metadata generated.

3.1 ECMAScript

JavaScript (JS) is a programming language mainly used in the development of client side web applications, also being one of the most popular programming languages. According to both GitHub and StakeOverflow statistics, JavaScript finished 2021 as second most active languages on GitHub[1] as well as on Stack-Overflow.[2]

ECMAScript standard[2] is the official document, written in the English language, in which the JavaScript language is defined. This document is in constant evolution, being updated by the ECMA Technical Committee 39 (TC39), which is responsible for maintaining the standard. The standard is currently in its twelfth version. The standard specifies the `JavaScript` language, to ensure its multiple compilers and interpreters implementations are coherent. Some of the `JavaScript` compilers are the Hop [1] and the JSC [5] compilers, the most popular interpreters are Node.js [**?**] and SpiderMonkey [3]. These are only four implementations among many others, which come along with the many use cases that `JavaScript` has. `JavaScript` is mostly used in the web context, both client-side within browsers and server-side, but also in embedded devices. Since `JavaScript` is used in so many scenarios and across so many different contexts, it is highly important that ECMAScript standard is defined in great detail to ensure consistency. Browsers, for example, need to run `JavaScript` implementations that coincide so that websites are correctly rendered and exhibit the same behavior. In order to achieve coherent implementations, the standard defines the types, values, objects, properties, syntax, and semantics of `JavaScript` that must be the same in every `JavaScript` compiler and interpreter, while allowing `JavaScript` implementations to define additional types, values, object, properties, and functions.

The `JavaScript` language can be divided into three major components, those being expressions and commands, built-in libraries, and finally internal functions.

- Expressions and commands describe the behavior of static constructions, detailing the semantics of the diverse expressions (e.g., assignment expressions,

[1] Second most utilized language based GitHub pull requests - https://madnight.github.io/githut/

[2] Tendencies based on the Tags used - https://insights.stackoverflow.com/trends

built-in operators, etc.), commands (e.g., loop commands, conditions command, etc.), and built-in types (Undefined, Null, Boolean, Number, String and Object).

- The internal functions of the language are used to define the semantics for both expressions and commands, as well as the built-in libraries. Internal functions are not exposed beyond the internal context of the language. In other words, no JavaScript program uses internal functions directly.
- Finally, built-in libraries encompass all the internal objects available when a JavaScript program is executed. Internal objects expose many functions implemented by the language itself, including functions to manipulate numbers, text, arrays, objects, amongst other things.

The remaining subsection provides a description of the three types of artifact described in the standard.

Semantics of IF statement Figure 1 shows a snippet of the ECMAScript standard description of the `IF` command. In order to evaluate `IF` commands with the shape:

```
if (Expression) Statement1 else Statement2
```

the language begins by evaluating the `Expression` storing the result in the variable `exprRef` (step 1). The previous step will be used as Boolean, therefore, the result of the previous step will be converted to a Boolean using the internal functions `ToBoolean` and `GetValue`, and having the result stored in the variable `exprValue` (step 2).A different `Statement` will be followed depending on `exprValue`. If `exprValue` has the value `true` the variable `stmtCompletion` will have the evaluation of the first `Statement` (step 3). Otherwise, the variable `stmtCompletion` will store the result of evaluating the second `Statement` (step 4). Finally, a `Completion` will be returned, if the `stmtCompletion` has non empty value then it will be returned, however, when the value is empty it will be replaced with undefined (step 5).

IfStatement : **`if`** **`(`** *Expression* **`)`** *Statement* **`else`** *Statement*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ! ToBoolean(? GetValue(*exprRef*)).
3. If *exprValue* is **true**, then
   a. Let *stmtCompletion* be the result of evaluating the first *Statement*.
4. Else,
   a. Let *stmtCompletion* be the result of evaluating the second *Statement*.
5. Return Completion(UpdateEmpty(*stmtCompletion*, **undefined**)).

Fig. 1. ECMAScript definition of an if-else statement

Semantics of the Pop function The Array built-in is an object as any other in JavaScript. The main difference is in its properties. Array Objects have a property `length` that contains the size of the array, as well as a property for each element of the array (from zero to `length` minus 1).

Figure 2 shows a simplified version of an array performing the pop function, where `(a)` and `(b)` are the before and after respectively. Before preforming `pop` `(a)`, the array has three properties `length`, `0`, and `1`. Property `length` represents the size of the array that has value `2`, while the properties `0` and `1` store the first (`banana`) and second (`kiwi`) elements of the array respectively. After `pop` is preformed `(b)`, the last element is of the array is removed (highlighted in red at `(a)`) and the `length` property (highlighted in green) is decremented by one since the size of the array changes to one.

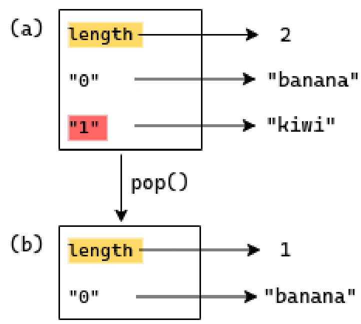

Fig. 2. Example Array.pop

Figure 3 shows a snippet of the ECMAScript standard description of the pop function in the Array Built-in. To begin with, the array will be converted to and Object using the `ToObject` function, and stored in the `O` variable (step 1). Afterwards, the array length of the previously calculated variable will be calculated with the `LengthOfArrayLike` internal function, and storing the result in the `len` variable (step 2). At this point there are to ways to proceed depending on the value of `len`. If the value is zero, the Array is empty, then the property `length` of `O` is set to zero and `undefined` is returned (step 3). Otherwise, when `len` is different from zero, meaning that the Array is not empty, the Array's last element will be removed (described in Figure 2) and returned (step 4). To begin with, the language will assert that `len` is positive (step 4.a). Afterwards, the `newLen` variable will store the value of `len` decremented by 1 (step 4.b). The variable `index` will store the variable calculated in the previous step represented as a String converted with the `toString` function (step 4.c). Then, stores the value of the `O` variable at the property corresponding to `index` in the `element` variable using the `Get` function (step 4.d). Subsequently, deletes the previously mentioned property of the `O` variable with the `DeletePropertyOrThrow` function (step 4.e). In addition, sets the `length` property of the `O` variable to the `newLen` using the `Set` function (step 4.f). Finally, returning the value of the variable `element` (step 4.g).

1. Let *O* be ? ToObject(**this** value).
2. Let *len* be ? LengthOfArrayLike(*O*).
3. If *len* = 0, then
   a. Perform ? Set(*O*, **"length"**, **+0**$_{\mathbb{F}}$, **true**).
   b. Return **undefined**.
4. Else,
   a. Assert: *len* > 0.
   b. Let *newLen* be $\mathbb{F}$(*len* - 1).
   c. Let *index* be ! ToString(*newLen*).
   d. Let *element* be ? Get(*O*, *index*).
   e. Perform ? DeletePropertyOrThrow(*O*, *index*).
   f. Perform ? Set(*O*, **"length"**, *newLen*, **true**).
   g. Return *element*.

Fig. 3. ECMAScript definition of Array.pop

LengthOfArrayLike internal function Figure 4 shows a snippet of the ECMAScript standard description of the `LengthOfArrayLike` internal function, that evaluates the function:

```
LengthOfArrayLike (obj)
```

The language starts by asserting that `obj` is an `Object` (step 1). Afterwards, gets the value of the property `length` from `obj` using the function `Get`. Then, converts the previously mentioned value to an Integer that represents the length with the `ToLength` function, and finally returns said Integer (step 2).

1. Assert: Type(*obj*) is Object.
2. Return $\mathbb{R}$(? ToLength(? Get(*obj*, **"length"**))).

Fig. 4. ECMAScript definition of the LengthOfArrayLike

3.2 Test262

Implementing a `JavaScript` engine is particularly difficult since it involves dealing with the many corner cases that exist in the language. To test that corner cases are correctly dealt with there is `Test262`[4], the ECMAScript standard test battery. Although, `Test262` is vital to the `JavaScript` engines, it is very hard to maintain due it's complexity, the total number of tests is around 39837 divided into 87 subfolders, each correspond to roughly one section of the standard. `Test262` complexity grows with changes to the standard since in most cases backward compatible is maintained except for a few select cases.

Due to the ECMAScript standard being so extensive most implementations are only partial, especially implementations and analysis developed in academic contexts. In order to test partial implementations, one must be able to obtain the applicable set of tests from all the tests contained in Test262. Selecting the applicable tests is not a trivial matter because there are too many tests and too many features. The current methodology is that each development team manually selects the tests that are applicable to their corresponding implementation. This raises the problem that there is not standard and precise way of picking the all the right tests from the almost 40000 tests in `Test262`, making the possibility of human error when selecting the applicable tests likely.

Figure 5 shows a test from `Test262`. Every test of `Test262` has 3 parts: first is the copyright section represented with the comment `//` (lines 1 and 2), second is the `Frontmatter` section between `/*---` and `---*/` with some metadata about the test (lines 4 to 7), and finally is the `Body` section with the code of the test (lines 9 to 13). The copyright section has information about the owner and license of the test. The `Frontmatter` section has the test id (15.4.5-1) and a description of the test. Finally, the `Body`'s code tests the correct implementation of the standard.

```
1   // Copyright (c) 2012 Ecma International.  All rights reserved.
2   // This code is governed by the BSD license found in the LICENSE file.
3
4   /*---
5   es5id: 15.4.5-1
6   description: Array instances have [[Class]] set to 'Array'
7   ---*/
8
9   var a = [];
10  var s = Object.prototype.toString.call(a);
11
12  assert.sameValue(s, '[object Array]',
13          'The value of s is expected to be "[object Array]"');
```

Fig. 5. Test262 es5id: 15.4.5-1

The `Frontmatter` has keywords to hold metadata of the test. These keywords are associated with specific elements of metadata concerning the test. Bellow is the list of possible keywords and their meaning:

- `description` - contains a short description about what will be tested;
- `esid` - contains the hash identifier of the ECMAScript portion associated with the feature that will be tested (the identifier references the most recent version of ECMAScript when the test is created);
- `info` - contains a deeper explanation of the test behavior, frequently includes a direct citation of the standard;
- `negative` - indicates that the test throws an error; associated to the keyword will be the type of error the test is supposed to be thrown (e.g. `TypeError`, `ReferenceError`) as well as the phase in which the error is expected to be thrown (e.g. `parse` vs `resolution` vs `runtime`);
- `includes` - contains the list of `harness` files that should be included in the execution of the tests (`Test262` makes use of a large number of auxiliary function defined in a dedicated library referred to as the `Test262 harness` described later in this section);
- `author` - contains the identification of the author of the test;
- `flags` - contains a list of booleans for each test property, the properties being: (1) `onlyStrict`, the test is only executed in strict mode; (2) noStrict, the test will only be executed in mode *sloppy*; (3) `module`, the test must be integrated as a `JavaScript` module; (4) `raw`, executes the test without any modification, which implies running as `noStrict`; (5) `async`, the test is contains asynchronous functions; (6) `generated`, the test generates the files specified by the property; (7) `CanBlockIsFalse` and (8) `CanBlockIsTrue`, the test will run if the property `CanBlock` of the `Agent Record` executing it is false and true respectively; (9) `non-deterministic`, indicates that the semantics used in the test are intentionally under-specified and therefore the test passing or failing should not be regarded as an indication of reliability or conformance;
- `features` - contains a list of features that are used in the test;
- `es5id` and `es6id` - indicates that the feature being tested belongs to ECMAScript 5 and 6 respectively and contains the hash identifier of the section of the standard it belongs to; these keywords have been deprecated and substituted by esid.

As Figure 5 illustrates, it is often the case that the metadata of a test is incomplete. Some tests also have the wrong metadata. As part of this thesis, we plan to process all the tests to check and correct their corresponding metadata as well as completing the metadata that is missing.

The example in Figure 5 has 2 keywords, description and the deprecated es5id. Besides the obvious upgrade from `es5id` to `esid` it would be useful to have `includes` with the `harness` files needed to execute the test. The `harness` information is very useful since it makes it easy to identify the part of the `harness` needed to run that test, opening the door for loading only part of the `harness` instead of the whole `harness` which is the current approach.

New Metadata In order to have a more complete `Frontmatter` we suggest adding the following information:

– `syntactic construct` - list of all syntactic constructions used in the test;
– `version` - the ECMAScript version of the standard in which the feature being tested was introduced;
– `built-ins` - list of all the built-ins used in the test.
– `harness-functions` - list of all the harness functions used to asses the test's results.

This metadata provides helpful information to solve the problem mentioned before, selecting the applicable tests for partial implementations, by allowing developers to filter tests by `builtin`, `static construct` and `version` of the `ECMAScript` standard. This would provide consistency and standardization to the selection of applicable tests to a partial implementation of the standard. As for the `harness-functions`, it provides the information of about the functions of `harness` that are used in the test, that is relevant because only a small part of the `harness` is need in each test even though the whole library is loaded.

3.3 An Infrastructure for testing reference implementations of the ECMAScript standard

This thesis continues the work done in the master thesis of Miguel Trigo titled *Infra-estrutura de Testes para Implementações de Referência do Standard ECMAScript*. In Miguel Trigo's master thesis, he processed all Test262 tests and checked their metadata, correcting some errors that were found and complementing some metadata that is missing in the `Frontmatter`. The author also added new metadata that out of the scope of the `Frontmatter`, added statistical data about the tests, and made all the previously mentioned data available in a `MongoDB` database.

Miguel Trigo processed the test in Figure 5 and generated the metadata in Figure 6. The `Frontmatter` in the test has the `es5id` and `description`, in the generated metadata they are represented for the `name` and `value` pairs `esid` and `description` respectively (updating the deprecated es5id). The following `names` were added by Miguel Trigo:

– `path` - stores the path to the test within `Test262`;
– `version` - stores the version of the `ECMAScript` standard that the test corresponds to;
– `built-ins` - stores a subpart of the `path`;
– `Array` - stores the subpart of the `path` that follows `built-ins`;
– `syntactic_construct` - stores an array with all the syntactic constructions used in the test;
– `builtIns` - stores an array with all the builtins used in the test;
– `asserts` - stores the number of times the `assert` library is used in the test;
– `error` - TODO Nao sei o que representa;
– `esprima` - stores whether or not the test is supported by `Esprima`[?] (`Esprima` is an standard-compliant `ECMAScript` parser that is also developed in `ECMAScript`);

– `lines` - stores the number of lines of code written in the test, ignoring the empty lines and comments (in this example the `value` is 3 because lines 12 and 13 in Figure 5 are one single line in the actual test in `Test262` that was split in the example to make the image more legible).

```json
{
    "path": "./test262-main/test/built-ins/Array/15.4.5-1.js",
    "version": 5,
    "esid": " 15.4.5-1",
    "description": " Array instances have [[Class]] set to 'Array'",
    "built-ins": "Array",
    "Array": "15.4.5-1.js",
    "syntactic_construct": [
        "Identifier",
        "ArrayExpression",
        "VariableDeclarator",
        "VariableDeclaration",
        "MemberExpression",
        "CallExpression",
        "Literal",
        "ExpressionStatement",
        "Program"
    ],
    "builtIns": {
        "Array": [],
        "Object": [
            "prototype",
            "prototype.toString"
        ],
        "Function": [
            "call"
        ]
    },
    "asserts": 1,
    "error": 0,
    "esprima": "supported",
    "lines": 3
}
```

Fig. 6. Metadata generated for test esid: 15.4.5-1.js

The metadata can be separated into four parts. Firstly, the `Frontmatter` part, `description` and `esid`, that stores information related to the `Frontmatter` of the test. Secondly, the path part comprised of `path`, `built-ins`, and `Array`, that contains the information related to the path of the test inside the `Test262`. Thirdly, the statistical part formed by `asserts`, `error`, `esprima`, and `lines`, that provides statistics on the code of the test and the metadata generation process. Finally, the new metadata part which consists of `version`, `syntactic_construct`, and `builtIns`, the new metadata is relevant information about the tests that should be added to the `Frontmatter`.

In the remaining of the subsection is concerned with how the new metadata, namely `syntactic_construct`, `builtIn`, and `version`.

Calculation of syntactic_Construct metadata The `syntactic_construct` metadata is calculated with the help of `Esprima`, which allows the analysis of the syntactic tree of the test. The tree analysis of the test is represented in a JSON format and the `syntactic_constructs` are the `values` corresponding to the `names` *type* of every element of the tree. This way, to get all the `syntactic_construct` is only a matter of traversing the tree and adding any new one found. The main problem being that `Esprima` only supports till version 7 of the `ECMAScript` standard, and also the `negative` tests that `Esprima` does not support.

```
Object.seal(new (Object.getPrototypeOf(() => {}).constructor)());
```

Fig. 7. Example test


```json
{
  "type": "Program",
  "body": [
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "CallExpression",
        "callee": {
          "type": "MemberExpression",
          "computed": false,
          "object": {
            "type": "Identifier",
            "name": "Object"
          },
          "property": {
            "type": "Identifier",
            "name": "seal"
          }
        },
        "arguments": [
          {
            "type": "NewExpression",
            "callee": {
              "type": "MemberExpression",
              "computed": false,
              "object": {
                "type": "CallExpression",
                "callee": {
                  "type": "MemberExpression",
                  "computed": false,
                  "object": {
                    "type": "Identifier",
                    "name": "Object"
                  },
                  "property": {
                    "type": "Identifier",
                    "name": "getPrototypeOf"
                  }
                },
                "arguments": [
                  {
                    "type": "ArrowFunctionExpression",
                    "id": null,
                    "params": [],
                    "body": {
                      "type": "BlockStatement",
                      "body": []
                    },
                    "generator": false,
                    "expression": false,
                    "async": false
                  }
                ]
              },
              "property": {
                "type": "Identifier",
                "name": "constructor"
              }
            },
            "arguments": []
          }
        ]
      }
    }
  ],
  "sourceType": "script"
}
```


Fig. 8. syntactic tree of the example test in Figure 7

Calculation of built-ins metadata The `built-ins` are calculated with two separate methods. The first method makes use of syntactic tree generated from `Esprima`, to find key `static_constructs` to identify the `built-ins` being used. While the second method goes through the code of a test and finds all the functions and associates them with their `built-ins`, as well as the finding `built-ins` being used directly. In both methods, after all the `built-ins` of a test are identified, the `built-in` that corresponds to the most recent version of the `ECMAScript` standard identifies the version of that test.

Calculation of Version metadata The `version` metadata is calculated by identifying the element, `built-in`, `syntactic_construct` or change in behavior of functions that is part of the newest version of the `ECMAScript` standard. The version is calculated using 3 different approaches dynamic, static, and mixed. The dynamic approach is based on the waterfall model, running the tests against an implementation of the standard. if the test passes then the test is associated with that version of the standard, otherwise the test will be run against the next version of the implementation of the standard. The static approach uses the syntactic tree generated with `Esprima`. From the analysis of the syntactic tree all the objects, functions, properties, operators, variables, and syntactical constructions present in the test are found, then searches for the oldest version that contains all of them. The mixed approach is calculated using the dynamic and static approaches. The mixed approach consists of running the results of the dynamic approach of each version and verifying that is does not use features introduced in a posterior version. The dynamic approach makes use of `SpyderMonkey` and `Node.js` implementations of the standard.

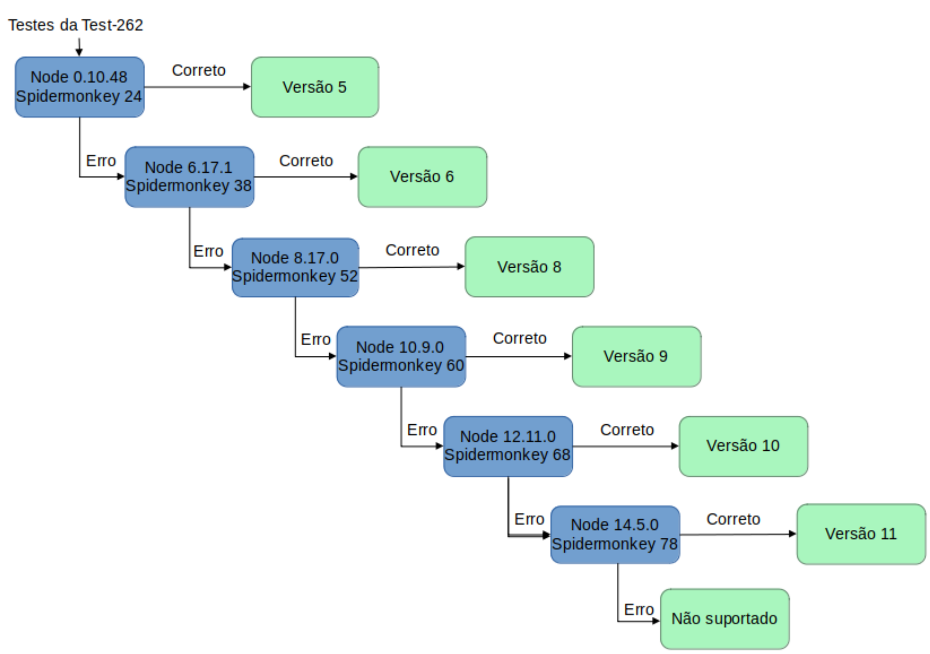

Fig. 9. waterfall model of the dynamic approach

4 Related Work

5 Design and Methodology

This thesis is a continuation of Miguel Tringo's master thesis, aiming to complete the calculation of the `Frontmatter` metadata especially the metadata about the `harness`, also aims to improve the generated metadata. Another aim of this thesis is to create a website for visualizing the metadata generated. Finally, we aim to build a platform that allows users to submit the markdown generated from executing the `Test262` and compare different runs.

Miguel Trigo's thesis metadata is incomplete in various ways, tests with unknown `version` around 9000, tests without `built-ins` around 17000, and tests without `syntactic_construct` around 13000 tests without esid around 600. The metadata from the thesis could be improved in way the data is arranged, for example, the subfolders of the path to the test are spread into the JSON Object of the test. The subfolders information being put into an array of ordered subfolders would increase the readability.

This thesis plans to improve the `version` metadata generation in two dimensions precision and efficiency. For the precision, with more `JavaScript` engines being used, there would be more certainty when determining the version. As for the efficiency, it would be possible to parallelize the waterfall model of the dynamic approach running multiple tests at the same time. It is also possible to use `git diff` to identify the tests that were added or changed since the last time the dynamic approach was executed, only needing to execute the dynamic approach on the differences.

The website for visualizing the metadata aims to make access to the metadata and filtering it easily accessible. The website is planned to allow the search

6 Evaluation and Planning

pedir descricao da teses de uma das primeiras reunioes (iPad)

7 Conclusion

References

1. Ambiente de programação multi-tier para o javascript, https://github.com/manuel-serrano/hop, acedido a 2020-12-21
2. especificação da linguagem ecmascript® , 6.0 edition / june 2015, http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf, acedido a 2020-12-21
3. Spidermonkey is mozilla's javascript engine written in c and c++, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, acedido a 2020-12-21
4. Test262 - official ecmascript conformance test suite, https://github.com/tc39/test262/, acedido a 2020-06-07
5. Um motor javascript que corre em cima do motor javascript do browser, https://github.com/mbbill/JSC.js, acedido a 2020-12-21

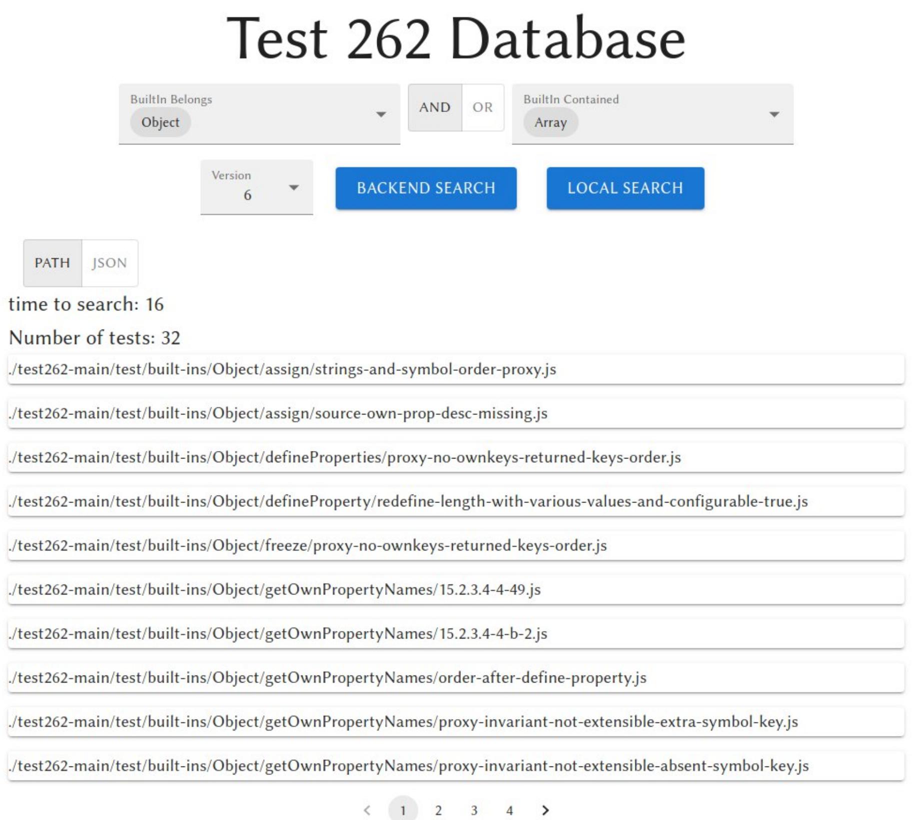

Fig. 10. Website for searching the metadata in construction