JSSpecVis: A JavaScript Language Specification Visualization Tool

Minseok Choe*
goro8pyo@korea.ac.kr
Korea University
Seoul, South Korea

Kyungho Song*
kh07019@gmail.com
Sogang University
Seoul, South Korea

Hyunjoon Kim
rickykhj@korea.ac.kr
Korea University
Seoul, South Korea

Jihyeok Park†
jihyeok_park@korea.ac.kr
Korea University
Seoul, South Korea

Abstract

ECMA-262 is the official language specification for JavaScript that defines the language semantics in detail. However, readers often struggle to understand the specification due to the intricate edge cases and lengthy, highly nested explanations. To address this issue, we present JSSpecVis, an interactive web interface visualizing the JavaScript language specification. For an intuitive understanding of edge cases, it provides *example programs* for each part of the specification with a *call-path context* selected by readers. In addition, it supports the *interactive execution* of JavaScript programs on the specification with a new debugging feature called *resume* and *provenance* for advanced debugging of JavaScript programs. Using these functionalities, JSSpecVis provides an intuitive learning environment for beginners and a powerful productivity for experts. Our tool is open at https://github.com/ku-plrg/js-spec-vis, and the tool demonstration is available at https://youtu.be/xqLPmVVlORQ.

CCS Concepts

• **Software and its engineering** → **Formal language definitions**; • **Human-centered computing** → **Visualization systems and tools**.

Keywords

JavaScript, Language Specification, Feature-Sensitive Coverage

ACM Reference Format:
Minseok Choe, Kyungho Song, Hyunjoon Kim, and Jihyeok Park. 2025. JSSpecVis: A JavaScript Language Specification Visualization Tool. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728579

1 Introduction

ECMA-262 [5] is the official language specification for JavaScript that defines the language semantics using pseudo-code algorithms consisting of structured steps written in English prose. It has been annually updated since 2015, and its latest version, ES2024, contains 816 pages with 2,773 algorithms consisting of 20,532 steps. There are three types of readers who refer to the specification to understand the language semantics for different purposes:

(1) **Language Designers** who design new language features and maintain the specification. The Technical Committee 39 (TC39) is the official committee who maintains ECMA-262.
(2) **Language Tool Developers** who implement the language tools. For example, developers of JavaScript engines (e.g., V8), transpilers (e.g., Babel), and static analyzers (e.g., ESLint).
(3) **JavaScript Programmers** who write JavaScript programs. They want to understand why and how JavaScript programs behave in a certain way to write programs correctly.

However, understanding the JavaScript language semantics with the specification is often painful for all three types of readers.

First, the most significant challenge is understanding the diverse and intricate *edge cases* in the language semantics. In particular, language designers and tool developers need to grasp all the edge cases to implement the tools and maintain the specification correctly. However, language features in JavaScript have a wide range of edge cases due to its highly dynamic nature with complex implicit type coercion rules. For example, the semantics of addition operators (+) contain more than 12 non-trivial edge cases that throw exceptions for different reasons. In addition, the semantics of different language features often share the same auxiliary algorithms. It leads to more challenges in understanding how edge cases in each auxiliary algorithm affect the semantics of different features.

Second, readers also struggle to follow lengthy and highly nested algorithm steps to understand the semantics. To understand the full semantics of a JavaScript program, readers need to follow lengthy steps across multiple algorithms, making it challenging to understand the semantics of even a simple expression. For example, readers need to follow 121 steps across 10 algorithms in the specification to understand why the JavaScript expression `1+1n` throws a `TypeError` exception. Besides, readers should keep track of not only the JavaScript program states but also the meta-level states of the specification when following the steps.

To address these challenges, we present JSSpecVis, an interactive web interface visualizing the JavaScript language specification. Our key idea is to provide 1) *example programs* related to each specification part in Program Visualizer and 2) *interactive execution* of JavaScript programs on the specification using Double Debugger. Example programs help readers intuitively understand edge cases

*Both authors contributed equally to this research.
†Corresponding author.

FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728579

Array.prototype.at (*index*)

1. Let *O* be ? ToObject(**this** value).
2. Let *len* be ? LengthOfArrayLike(*O*).
3. Let *relativeIndex* be ? ToIntegerOrInfinity(*index*).
4. If *relativeIndex* ≥ 0, then
   a. Let *k* be *relativeIndex*.
5. Else,
   a. Let *k* be *len* + *relativeIndex*.
6. If *k* < 0 or *k* ≥ *len*, return **undefined**.
7. Return ? Get(*O*, ! ToString(ℝ(*k*))).

Figure 1: An algorithm for `Array.prototype.at` in ES2024

in language semantics. Program Visualizer is an interactive web interface, available as a Chrome extension, which provides two types of example programs for each specification part: 1) synthesized minimal programs and 2) official conformance tests from Test262 [6]. Readers can select a *call-path context*, a sequence of call-sites in the algorithms, to see more specific example programs under the selected context. Double Debugger supports interactive execution of JavaScript programs on the specification. It provides basic debugging features, shows states of both the JavaScript program and the specification during the execution, and supports novel debugging features: *resume* and *provenance*. Users can *resume* the execution of the minimal program starting from the selected step and call-path context in the visualizer. Moreover, it keeps track of the *provenance* (i.e., the allocation site) of each record and allows users to step-back to its provenance to show the origin of values or exceptions.

In the remainder of this paper, we first introduce the challenges in understanding the JavaScript language specification (§2). Then, we present the design and implementation of JSSpecVis (§3), discuss the related work (§4), and conclude the paper (§5).

2 Background and Motivation

This section explains the basic notations in the JavaScript language specification and the challenges in understanding the specification.

ECMA-262 is the official language specification for JavaScript and defines language semantics using pseudo-code algorithms for each syntactic feature or built-in function. Figure 1 shows an algorithm for the `Array.prototype.at`[1] built-in function in ES2024, which retrieves an element from any array-like object at the given index. This feature was introduced in ES2022 and supports negative indices as well. For instance, `[3, 7, 11].at(1)` returns the second element `7`, and `[3, 7, 11].at(-1)` returns the last element `11`. The algorithm consists of seven steps to describe its semantics with the help of auxiliary algorithms (e.g., ToObject and LengthOfArrayLike).

In the specification, each branch often denotes an edge case in the semantics. For example, the step 5.a in Figure 1 handles the negative index case, and the step 6 handles out-of-bound indices. In addition, the ? operator is another kind of branch for abrupt completions. In the specification, a *completion record* is a special type used to explain runtime propagation of values or control flows, such as `break`, `return`, or exceptions. It is not a JavaScript value and only exists in the specification. There are two kinds of completions: normal and abrupt. If a completion is normal, it captures the produced value;

Array.prototype.at (*index*)

2. Let *len* be ? LengthOfArrayLike(*O*).

LengthOfArrayLike (*obj*)

1. Return ℝ(? ToLength(? Get(*obj*, **"length"**))).

ToLength (*argument*)

1. Let *len* be ? ToIntegerOrInfinity(*argument*).

ToIntegerOrInfinity (*argument*)

1. Let *number* be ? ToNumber(*argument*).

ToNumber (*argument*)

10. Return ? ToNumber(*primValue*).

Figure 2: A path from `Array.prototype.at` to ToNumber that throws a `TypeError` exception in ES2024

otherwise, it captures the abrupt reason. The ? operator checks if the given value is an abrupt completion and directly returns it as the result of the algorithm. Otherwise, it takes the captured value from the normal completion. The `!` operator is similar to this, but it assumes the given value is a normal completion. Thus, readers need to understand four ? operators in the algorithm to understand the full semantics of `Array.prototype.at`.

However, understanding the edge cases is often challenging even for experienced readers. For example, the `core-js` polyfill library is essential for uniform JavaScript execution environments, but its developers introduced a bug[2] by misunderstanding the edge case for the ? operator in the step 2 of the algorithm in Figure 1. As a result, the following JavaScript program returns `undefined` value when using the `core-js` library, even though it should throw a `TypeError` exception according to the specification:

```
// TypeError in ES2024, but undefined in core-js v3.35.0
Array.prototype.at.call({ length: -1n }, 0);
```

One contributing factor to this bug is the absence of a test case addressing this edge case in Test262 [6], the official conformance test suite. The coverage information about which edge cases in each language feature Test262 covers is crucial for both test suite maintainers and tool developers.

In addition, auxiliary algorithms have their own edge cases and are used in multiple places in the specification. Thus, readers need to understand how their edge cases affect the semantics of its caller algorithms. For example, the LengthOfArrayLike algorithm is called in 55 different places in the ES2024 specification and has *two* different own edge cases. It means that the edge case for the step 2 of the `Array.prototype.at` can be triggered by two different edge cases in the LengthOfArrayLike algorithm. Similarly, the semantics of the LengthOfArrayLike algorithm is affected by the ToLength and Get algorithms, which have their own edge cases as well.

A deep path across multiple algorithms makes it difficult to understand the language semantics as well. The precise reason of the above bug in the `core-js` library is related to the branch in the step 2 of the ToNumber algorithm through the path across five algorithms from the `Array.prototype.at` algorithm as depicted in Figure 2. Besides, readers need to keep track of the states of both the JavaScript program and the specification when following the

[1] https://tc39.es/ecma262/2024/#sec-array.prototype.at

[2] https://github.com/zloirock/core-js/issues/1318#issue-2063353379

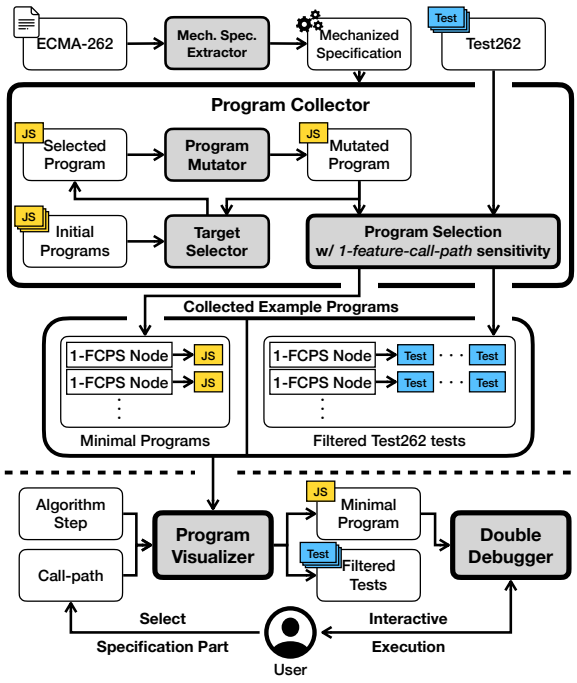

Figure 3: Overall structure of JSSpecVis

steps, which makes it even more burdensome to understand the language semantics correctly.

3 Tool Design and Implementation

As depicted in Figure 3, JSSpecVis consists of three components: 1) Program Collector as a preprocessor, 2) Program Visualizer for browsing the collected programs, and 3) Double Debugger for interactive execution of JavaScript programs on the specification. We extend the ESMeta toolchain [1, 18] to extract the mechanized specification and use it for all three components. A *mechanized specification* is a program written in IR_{ES}, an intermediate representation of ECMA-262, and executable as a JavaScript interpreter to simulate the execution of JavaScript programs on the language specification. This section explains the design and implementation of each component of our tool in detail.

3.1 Program Collector

It collects two types of example programs for each node in the mechanized specification: 1) minimal JavaScript programs and 2) official conformance tests from Test262. We first explain the concept of feature-sensitive coverage [16] on the mechanized specification, and then describe how we synthesize minimal programs and select example programs using the coverage information.

3.1.1 Feature-sensitive Coverage. A *feature-sensitive (FS) coverage* criterion is an extension of a graph coverage criterion that splits coverage targets with their innermost enclosing language feature. For example, if a node in the control-flow graph of the mechanized specification is reachable from two different language features (e.g., + and – operators), the coverage criterion considers the node as two separate coverage targets for each feature. It is possible to further

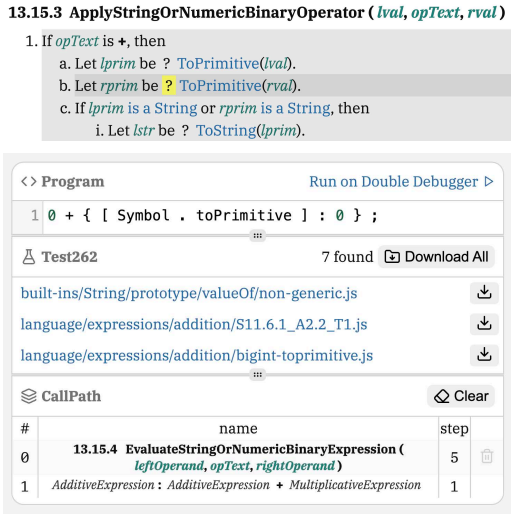

Figure 4: A minimal program and Test262 tests under the selected call-path context in Program Visualizer

extend the FS coverage with k innermost enclosing features, called k-FS coverage. Furthermore, k-feature-call-path-sensitive (k-FCPS) coverage is an extension of k-FS coverage that considers not only features but also the call path from the feature to the target node to split coverage targets. In this work, we use 1-FCPS node coverage in the specification because we want to show example programs under the selected call-path context in the program visualizer.

3.1.2 Synthesizing Minimal JavaScript Programs. Our tool synthesizes minimal JavaScript programs using mutation-based fuzzing. It starts with the seed programs consisting of 9,092 programs collected from our previous study [16]. Target Selector randomly selects a program from the seed programs and Program Mutator mutates the program to generate new programs. We use five mutation methods: 1) random mutation, 2) random removal, 3) nearest syntax tree mutation, 4) statement insertion, and 5) string substitution. In 30 hours, our tool synthesizes 10,047 minimal programs that cover 324,518 1-FCPS nodes in the mechanized specification.

3.1.3 Program Selection with 1-FCPS Coverage. We select example programs for each 1-FCPS node in the mechanized specification using the coverage information. For each 1-FCPS node, we select 1) a minimal program and 2) a set of official conformance tests that touch the coverage target. Since conformance tests utilize harness functions as helpers, we ignore the touching information for coverage targets when the innermost enclosing syntax tree node is in the harness functions. As a result, we collect 10,047 minimal programs and 25,276 official conformance tests from Test262.

3.2 Program Visualizer

Program Visualizer provides an interactive web interface for browsing the collected example programs. It is implemented as a Chrome extension that extends the online ECMA-262 page. Users can select interesting parts of the specification by clicking on steps (or ? operators) and see the minimal JavaScript programs and conformance tests that cover the selected parts.

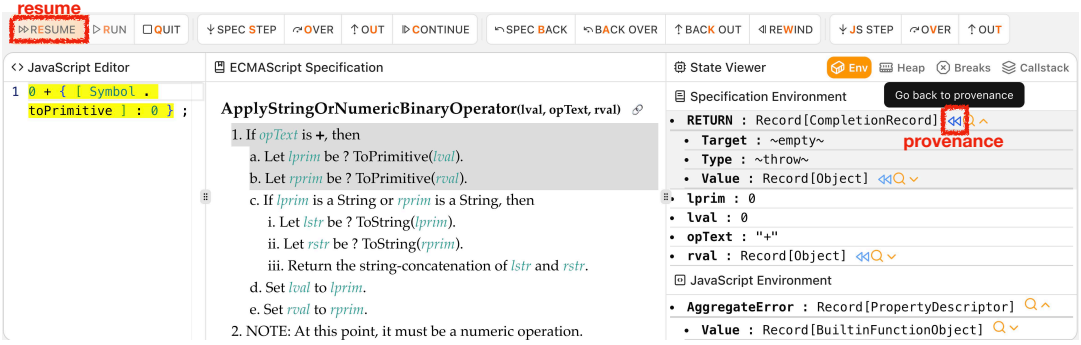

Figure 5: Double Debugger for interactive execution of JavaScript programs on the JavaScript language specification

In addition, if users move to other auxiliary algorithms to see the detailed semantics of features by clicking call-sites, it saves the call-path and uses it as a context to filter example programs. Thus, if users select a step under a call-path context, it automatically filters 1) 1-FCPS nodes matched with the selected step and 2) feature-call-paths ending with the selected call-path. Then, it shows example programs mapped from the filtered 1-FCPS nodes. We remove cycles when a user clicks already visited call-sites in the call-path to prevent infinite length of paths. If a user selects an infeasible call-site, it clears the call-path and starts a new context.

For example, Figure 4 shows the example programs related to an edge case in the ApplyStringOrNumericBinaryOperator algorithm in the context of the addition + operator feature. In this case, the user first selects the call-path context by clicking 1) step 1 of the + operator feature and 2) step 5 of EvaluateStringOrNumericBinaryExpression. Under this selected context, if the user clicks the ? operator at step 1.b of ApplyStringOrNumericBinaryOperator, the visualizer shows a minimal program and the 7 conformance tests. While 26 conformance tests exist for the ? operator at step 1.b, it shows only 7 tests related to the selected call-path context.

3.3 Double Debugger

Charguéraud et al. [3] introduced the concept of *double debugger* with JSExplain, that shows states of both the program and the JavaScript reference interpreter. However, it targets an interpreter written in OCaml rather than the specification and only supports ES5.1 version and requires manual update for later versions. Instead, we extract a mechanized specification from the latest ECMA-262.

Our Double Debugger provides specification-level navigation for algorithm steps (e.g., forward-step, backward-step, breakpoints, and continue) and also JavaScript-level navigation to focus on program execution. The call-stack tab highlights dependent parts of call-sites that affect the current state by calculating the intersection of dynamically executed and statically control/data dependent parts.

In addition, it provides two novel debugging features: 1) *resume* and 2) *provenance*. A user can execute provided minimal programs by clicking the **Run on Double Debugger** button in the visualizer and *resume* its execution from the selected context by clicking the **Resume** button in the debugger. Internally, it passes the recorded iteration count for the selected context to the debugger to resume the execution from the selected context. It helps users to understand the semantics of the selected part by the step-by-step execution in the debugger. The debugger keeps track of the provenance (i.e., allocation site) of each record in the specification and allows users to step-back to there, providing the origin of values or exceptions.

For example, a user can execute the debugger with the minimal program in Figure 4 by clicking the **Run on Double Debugger** button. Then, a user can *resume* the execution from the selected context by clicking **Resume** button in the debugger as shown in Figure 5. It shows the current return value is an abrupt completion. If a user clicks the provenance button (i.e. double-left arrow), the debugger steps-back to its provenance and shows that a `TypeError` exception is thrown at the step 3 of the GetMethod algorithm.

4 Related Work

We are inspired by live programming [8, 10, 19, 20], providing visualization for better understanding of programs. To visualize language specifications with examples, we need an executable specification. While diverse researchers [2, 4, 7, 9, 11] formalized JavaScript semantics, they mainly focused on ES5.1 and required manual updates. In contrast, ESMeta automatically extracts an executable specification for the latest ECMA-262 [15] and supports diverse extensions [12–14, 16]. Thus, we developed JSSpecVis on top of it.

One of the closest works to our approach is MetaData262 [17], which filters Test262 tests by language features. However, it requires manual metadata updates and lacks fine-grained selection, such as steps or edge cases in a single feature. Another one is JSExplain [3] that introduced a double debugger for JavaScript but relies on an OCaml-based interpreter for ES5.1, requires manual updates, and does not support debugging features like *resume* and *provenance*.

5 Conclusion

We present JSSpecVis to help users understand the JavaScript language specification by visualizing the specification with examples and interactive execution. Its debugging features support both beginners and experts in exploring JavaScript semantics effectively.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.RS-2024-00344597) and the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains)

References

[1] 2024. *ESMeta: An ECMAScript specification metalanguage used for automatically generating language-based tools.* https://github.com/es-meta/esmeta

[2] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* https://doi.org/10.1145/2535838.2535876

[3] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *Companion Proceedings of the The Web Conference (WWW).* https://doi.org/10.1145/3184558.3185969

[4] Noé De Santo, Aurèle Barrière, and Clément Pit-Claudel. 2024. A Coq Mechanization of JavaScript Regular Expression Semantics. *Proc. ACM Program. Lang.* 8, ICFP, Article 270 (Aug. 2024), 29 pages. https://doi.org/10.1145/3674666

[5] ECMA International. 2024. *ECMA-262, 15th edition, ECMAScript ®2024 Language Specification.* https://tc39.es/ecma262/2024

[6] ECMA International. 2024. *Test262: A conformanece test suite for the latest draft of ECMAScript.* https://github.com/tc39/test262

[7] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienundefined, Thomas Wood, and Philippa Gardner. 2017. JaVerT: JavaScript verification toolchain. *Proc. ACM Program. Lang.* 2, POPL, Article 50 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158138

[8] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST).* 737–745. https://doi.org/10.1145/3126594.3126632

[9] Adam Khayam, Louis Noizet, and Alan Schmitt. 2021. JSkel: Towards a Formalization of JavaScript's Semantics. In *JFLA 2021-Journées Francophones des Langages Applicatifs.* 1–22.

[10] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290327

[11] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: a complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15).* Association for Computing Machinery, New York, NY, USA, 346–356. https://doi.org/10.1145/2737924.2737991

[12] Jihyeok Park, Seungmin An, and Sukyoung Ryu. 2022. Automatically Deriving JavaScript Static Analyzers from Specifications Using Meta-Level Static Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* https://doi.org/10.1145/3540250.3549097

[13] Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu. 2021. JSTAR: JavaScript Specification Type Analyzer using Refinement. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* https://doi.org/10.1109/ASE51524.2021.9678781

[14] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 13–24. https://doi.org/10.1109/ICSE43902.2021.00015

[15] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 647–658. https://doi.org/10.1145/3324884.3416632

[16] Jihyeok Park, Dongjun Youn, Kanguk Lee, and Sukyoung Ryu. 2023. Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations. *Proc. ACM Program. Lang.* 7, PLDI, Article 126, 23 pages. https://doi.org/10.1145/3591240

[17] Frederico Ramos, Diogo Costa Reis, Miguel Trigo, António Morgado, and José Fragoso Santos. 2023. MetaData262: Automatic Test Suite Selection for Partial JavaScript Implementations. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023).* Association for Computing Machinery, New York, NY, USA, 1499–1502. https://doi.org/10.1145/3597926.3604923

[18] Sukyoung Ryu and Jihyeok Park. 2024. JavaScript Language Design and Implementation in Tandem. *Commun. ACM* (apr 2024), 13 pages. https://doi.org/10.1145/3624723 Online First.

[19] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE).* 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[20] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems?. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '97).* Association for Computing Machinery, New York, NY, USA, 258–265. https://doi.org/10.1145/258549.258721