POLITECNICO DI TORINO
Repository ISTITUZIONALE

IntelliGame in Action: An Experience Report on Gamifying JavaScript Unit Tests

Publisher copyright

(Article begins on next page)

22 December 2024

IntelliGame in Action: An Experience Report on Gamifying JavaScript Unit Tests

Philipp Straubinger*
University of Passau
Passau, Germany

Tommaso Fulcini*
Politecnico di Torino
Torino, Italy

Gordon Fraser
University of Passau
Passau, Germany

Marco Torchiano
Politecnico di Torino
Torino, Italy

ABSTRACT

This paper investigates the integration and assessment of IntelliGame, a gamification plugin initially designed for Java development, within the realm of JavaScript unit testing. We aim to verify the generalizability of IntelliGame to JavaScript development and to provide valuable insights into the experiment's design. For this, we first customize IntelliGame for JavaScript, and then conduct a controlled experiment involving 152 participants utilizing the Jest testing framework, and finally examine its influence on testing behavior and the overall developer experience. The findings from this study provide valuable insights for improving JavaScript testing methodologies through the incorporation of gamification.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Integrated and visual development environments**.

KEYWORDS

Gamification, IDE, IntelliJ, Software Testing

1 INTRODUCTION

IntelliGame [8] is a plugin for the popular IntelliJ IDEA, which enables gamification covering various aspects of software testing, such as test execution, coverage evaluation, debugging, and test refactoring. It introduces 27 different achievements, each with incremental levels, providing positive feedback when developers exhibit commendable testing behavior. IntelliGame provides real-time

*Both authors contributed equally to this research.

analysis of developer interactions, notifications, and a user interface within IntelliJ to display achievements and progress.

In its original evaluation [8], we conducted a controlled experiment with 49 participants to study the impact of IntelliGame on testing behavior. The experiment involved a Java programming task with participants divided into different groups, aiming to assess the impact of the gamified environment on testing behavior, test suites, achievement levels, code functionality, and the overall developer experience. The results showed clearly that IntelliGame had a positive effect on testing behavior. Participants using IntelliGame wrote more tests, achieved higher code coverage and mutation scores, ran tests more frequently, and implemented functionality earlier compared to the *control* group. The impact on resulting test suites was substantial, and achievements correlated positively with various testing metrics.

The goal of this paper is to build upon these promising results achieved with IntelliGame for Java environments, and to assess whether these outcomes can be replicated on a larger scale and in a different context. In particular, the extension of the positive impact of achievements in the IDE to various programming languages remains unexplored. Thus we shifted the focus of this study to JavaScript (JS), recognized as the most used language as of 2023 [6]. With the growing importance of JS, there is an increasing demand to instruct and practice language-specific testing activities. Expanding beyond the previous sample, this study engages a broader audience with multiple tasks, allowing for more extensive code development and testing. This expansion, coupled with the shift to a different programming language, considers both generalizability and effectiveness beyond a short-term context.

The objective of this paper is to provide an in-depth description of the experiment's design, including challenges encountered, lessons learned, and recommendations for future iterations of JS-based experiments. Notably, there is a gap in the existing literature regarding programming experiments centered on an open-source JavaScript target [9], emphasizing a fully replicable and systematic approach. In Section 2 we provide some background about the topic of the paper with also some examples of related works, inSection 3 the implementation approach for the JavaScript version of IntelliGame is described, while Section 4 shows the selection process for the project as a subject is meticulously detailed, shedding light on the rationale behind the choice.

Philipp Straubinger, Tommaso Fulcini, Gordon Fraser, and Marco Torchiano

2 BACKGROUND AND RELATED WORK

Compared to writing code, testing is often viewed as less rewarding by software developers, requiring significant effort without always resulting in due recognition from management [7]. To address issues related to motivation and acknowledgment of testers' contributions, various solutions have been proposed, encompassing both extrinsic and intrinsic motivators. Gamification, a rising trend in this domain, has experienced substantial growth over the past decade [3], even leading to dedicated conferences in the academic sector.

Defined as *the use of game elements in a non-playful context* by Deterding et al. [2], gamification involves creating a playful environment in which users can accomplish their daily tasks. Completing daily goals within such an environment allows users to benefit from the motivators associated with the designed game elements, making the underlying activity more engaging and satisfying.

Software testing has been successfully gamified in particular in the context of education. For example, Code Defenders [5] is a gamified application to teach mutation testing concepts in an academic setting, where students assume either of two roles: attackers or defenders. Users from both sides engage in a mutual challenge on a shared Java class, applying mutation testing. The attackers' goal is to create code mutants (variants of the class with the same functionality), while the defenders' objective is to enhance the existing test suite with new test cases to detect the code mutants. The gamified platform incorporates mechanics and dynamics that integrate social interaction, creativity, and competition.

Gamification has also been applied in a practical rather than educational context. For example, Coppola et al. [1] propose a framework to gamify exploratory GUI testing using a Capture and Replay tool for web applications. The gamification layer introduces common game elements such as scores and leaderboards, as well as novel elements not previously explored in the literature, including dynamic visual bug injection with related visual feedback and scores, along with a progress bar tracking the tester's exploration, i.e., the achieved widget coverage of a specific web page.

As indicated in a recent survey on gamified software testing [3], current trends in the literature show that unit testing is the most targeted level, with the most focused testing phase being test creation and execution. Popular techniques considered by researchers and practitioners include Mutation, Black Box, and White Box testing. While INTELLIGAME initially emerged as a unit test tool, its achievement-based structure allows for extensibility across various testing dimensions, introducing achievements specific to each.

3 IMPLEMENTATION

To enable JavaScript support in INTELLIGAME, we transitioned from using IntelliJ Community Edition to IntelliJ Ultimate[1]. This shift was necessary because JavaScript support is exclusively available in the latter. While we managed to adapt most of the existing test achievements for JavaScript, some proved challenging to port, and a few were even deemed impossible.

Adjusting for test execution tracking, such as the number of tests and assertions, was straightforward, thanks to the Jest testing framework's syntax[2]. However, obtaining coverage information

---

[1]https://www.jetbrains.com/products/compare/?product=idea&product=idea-ce
[2]https://jestjs.io/
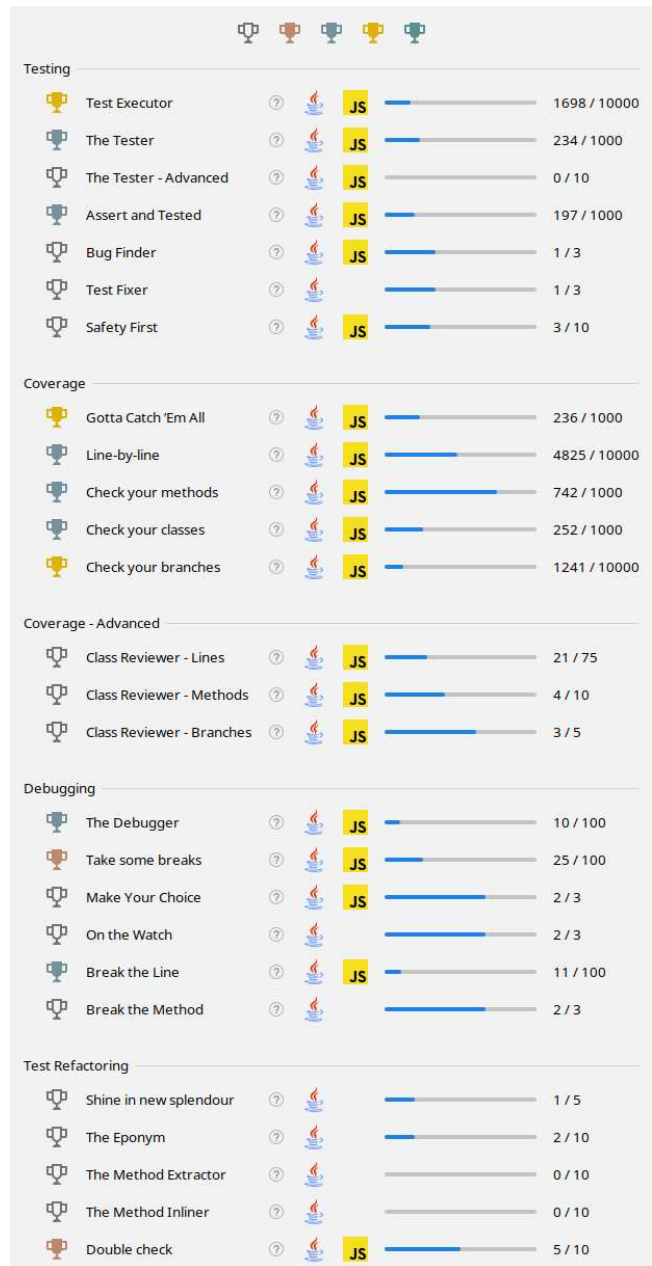
Figure 1: IntelliJ window showing achievements and progress

posed a greater challenge. The built-in test engine of IntelliJ lacks the same coverage details for JavaScript as it does for Java, such as covered branches or methods.

To address this, we devised a specialized test execution configuration that utilizes Jest to output overall coverage information to the console. We extract and employ this information for our achievements. Additionally, Jest writes coverage details to a JSON file, offering comprehensive class-specific information that we leverage for the remaining achievements. In summary, we extract JavaScript

coverage information from three sources: the built-in test engine, the console, and summary files.

Another obstacle arose with the JavaScript debugger, which lacks support for method breakpoints and field watchpoints, required by the implementation of some of the achievements. Unfortunately, we could not find suitable alternatives for these features. Similarly, most of the refactoring-related achievements, implemented using RefactoringMiner for Java[3], lacked equivalent tools for JavaScript.

Despite facing challenges, we successfully transferred 19 achievements out of 26 from Java to JavaScript. These achievements can be identified by a small JavaScript symbol (see Fig. 1). Each achievement is represented by a trophy indicating the level, a progress bar showing the current progress towards the next level, and a description detailing the progress required for reaching the next level, accessible by hovering over the question mark. For further details and explanations of the achievements, refer to related work [8].

4 EXPERIMENT

The primary aim of the experiment was to replicate the conditions outlined in the original validation paper [8], adhering to the guidelines set by Jedlitschka et al. [4]. However, as the goal was not to assess the effectiveness of the tool but rather to test its applicability with a different programming language and subjects, we introduced some modifications to the experimental approach.

4.1 Participant Selection

We invited graduate students from the Software Engineering course at Politecnico di Torinoto participate voluntarily in the experiment. Successful participation earned students additional points for their Software Engineering exam, with the assurance that their performance would not impact their bonus points to avoid bias. All participating students possessed prior JavaScript testing experience, as the Software Engineering course extensively covered unit testing using the Jest framework.

The sample of participants in this study comprised 152 individuals, with 85% identifying as male and the remaining 15% as female. Over 90% fell within the age range of 22 to 25, being graduate students. The participants were randomly assigned to one of two groups: *treatment* and *control*.

The majority of participants had less than three months of experience with JavaScript, and it is worth noting that the *treatment* group had less experience compared to the *control* group with 59% and 47% having less than three months of experience, respectively. Additionally, 96% of participants in both groups had less than three months of experience with Jest. In fact, the majority of participants were introduced to JavaScript and Jest during the course's lectures.

4.2 Project Selection

Initially, with a shift in the reference programming language, the need arose to select an alternative project, departing from the *FixedOrderComparator* Java class used in the original study [8].

Our criteria for project selection involved finding a subject that presented both challenge and feasibility within a 150-minute timeframe. We sought a real existing project to capture the complexity of a real-world scenario, emphasizing documentation, a comprehensive test suite, and modularity. Modularity was crucial, allowing participants to choose among modules, and deciding whether to tackle more complex or simpler functions.

The selection process began by exploring JavaScripting[4], a repository of publicly available JavaScript projects. Given the extensive collection, including popular frameworks like React, React Native, Vue, and Angular, we narrowed down our search to importable lightweight libraries with standardized functionalities. We focused on the *miscellaneous* category for a subject meeting our criteria.

The first project suiting our needs was *Date Fns*[5], an open-source project featuring 244 functions related to the date data type. Functions ranged from trivial tasks like determining the order between two dates to more complex operations like formatting data with ISO or RFC representation. The project, widely used (over two million times) and well-maintained by 300+ contributors with over a thousand forks, met our requirements.

The project, however, was developed in TypeScript (TS) rather than JS. Notwithstanding the difference between the two programming languages, given the high project usage and reliability, the full interoperability of JS code in TS and the possibility of directly transposing TS code into JS, was deemed that this language difference was not a reason for exclusion. Moreover, we consider it valuable for the reader to discuss in section 5.2 the problems encountered and the solutions put in place to resolve them.

For these reasons, we used the JS version obtained through transpilation [10]. The seamless interoperability between the two languages facilitated the conversion of functions from TS to JS, made possible with the help of the open-source automatic tool transform[6].

4.3 Experiment Task

We carefully chose a set of 23 functions from the original GitHub *Date Fns* project, prioritizing simplicity in terms of complexity and lines of code. The aim was to ensure that most students could confidently build and test these functions without feeling overwhelmed by their difficulty.

Before assigning the experiment task, we conducted a pilot study with three master's students nearing graduation. This helped validate the task's scalability. While no data was collected from them, the pilot ensured proper calibration by adjusting the assignment's complexity, excluding the three most challenging functions.

The functions' bodies were removed, leaving the requirements, function names, and parameters intact. The project incorporated complete JavaScript documentation for built-in date functions, eliminating the need for online references. Two verification methods were provided: a Jest configuration with empty test files for participants to complete and run, and a *main.js* file acting like the Main class in Java, mirroring the original experiment's setup [8].

Two sessions were held: the first with 60 participants and the second with the remaining individuals. This division allowed potential data discarding from the first day if any issues arose. Participants were split into a *control* group, tackling tasks without achievements, and a *treatment* group, completing tasks with the plugin enabled.

---

[3]https://github.com/tsantalis/RefactoringMiner

[4]https://www.javascripting.com/
[5]https://github.com/date-fns/date-fns
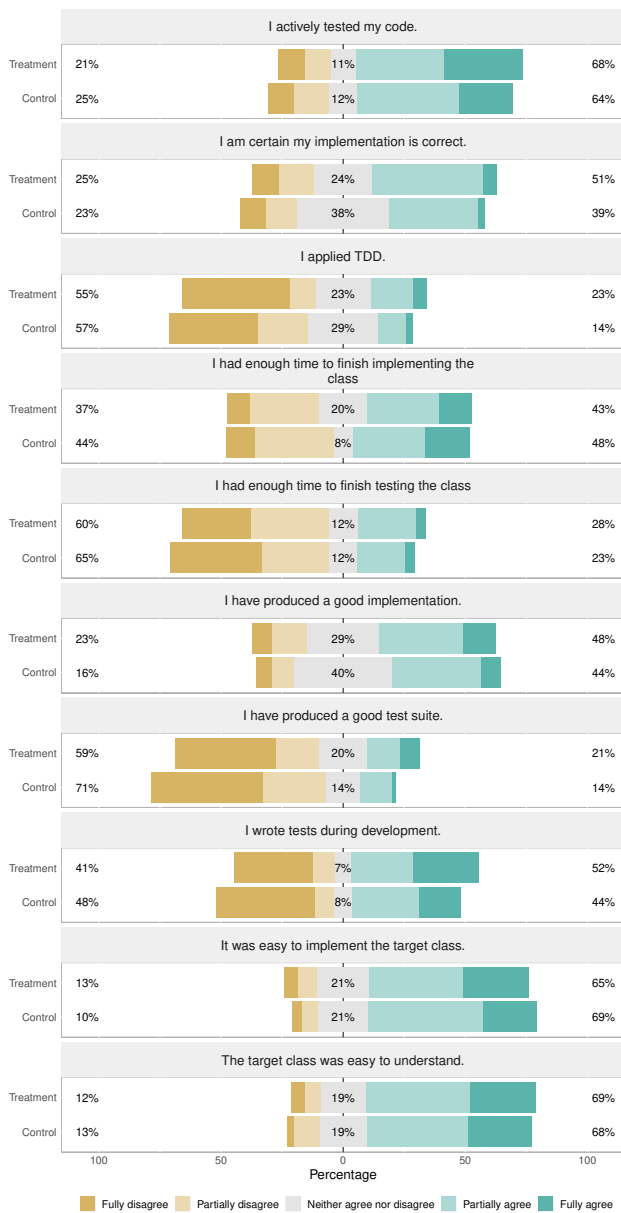[6]https://github.com/ritz078/transform

Figure 2: General survey responses – ranging from negative on the left to positive on the right

Participants from the *treatment* group were instructed on how the gamification plugin works but were not shown in detail the conditions for unlocking each achievement. These were shown in an overlay box displayed on the screen when hovering the mouse over each achievement of the list.

During the experiment, a custom event logger recorded achievement states after each user interaction. An automated script committed and pushed current implementations and log files to a Git repository every minute.

Following the 150-minute implementation phase, an exit questionnaire featured general questions for both groups. The treatment group received an additional page at the end of the survey inquiring about their plugin experiences, with answers on a five-point Likert scale. An optional free-text question allowed participants to share individual comments and feedback.

5 RESULTS

While the analysis of results is ongoing, we provide a preliminary overview by addressing the challenges encountered, our problem-solving approaches, and insights from the exit survey completed by students regarding their perceived user experience.

5.1 Survey Answers

Based on the responses to the exit survey (Fig. 2), the chosen target class was well-received by all groups. All groups unanimously agreed that the class was easy to comprehend and implement. However, it is noteworthy that just under half of the participants in the *treatment* group and fewer than 40% in the *control* group indicated they had sufficient time to complete the implementation. Consequently, it is not surprising that only approximately 25% in both groups felt they had enough time to conduct thorough testing of their implementations.

Interestingly, around two-thirds of all participants actively tested their code. Notably, a smaller percentage of participants wrote tests during the development phase, with 52% in the *treatment* group compared to 44% in the *control* group. Roughly 50% of participants in both groups were confident in the correctness of their implementations. Conversely, only 21% in the *treatment* group and 14% in the *control* group were certain about the quality of their test suites. No significant differences were observed between the two groups.

Concerning the responses of the *treatment* group regarding INTELLIGAME, they are predominantly positive or, at worst, undecided (Fig. 3). Participants demonstrated a clear understanding of the tool's descriptions and how to make progress in the presented achievements. They also appreciated the frequency of the notifications. About 40% of the participants reported that the achievements positively influenced their testing behavior, and an equivalent percentage mentioned being motivated by both the notifications and the plugin itself. Encouragingly, 42% of participants expressed a desire to use INTELLIGAME in their own projects.

5.2 Problems Faced and Lessons Learned

In the initial adaptation phase, we faced challenges while transitioning the original TypeScript project to JavaScript. Although automatic transpilation facilitated obtaining the correct JavaScript code, the hurdle lies in converting the TS configuration to a JS-compatible one. Since the original project used another testing framework, i.e., Karma, we had to modify the configuration to fit Jest.

Another challenge arose when configuring the *main.js* file for manual function assessment. Unlike Java, where a Main class defines the starting point, JavaScript, especially in the Node framework, lacks that. So, in order to reproduce the same setting as the original validation experiment [8], we had to create a custom file for this purpose.
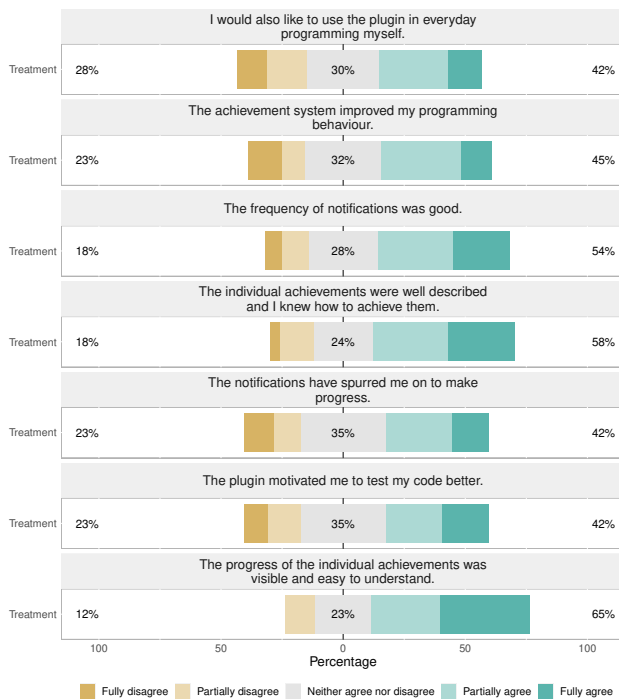
Figure 3: Answers regarding the IntelliGame plugin – ranging from negative on the left to positive on the right

Throughout the experiment, additional issues surfaced and required adaptation of the plugin. In the first session, participants could only choose between running the project via tests or main execution due to the provided configuration. Despite being shown how to switch between configurations, most participants stuck with one method for the entire session.

Upon reflection, we observed that the number of tasks overwhelmed participants, leading them to feel demotivated despite knowing that their performance would not be evaluated. While they found the functions appropriately challenging, the abundance of tasks influenced their survey responses negatively (e.g., rating their implementation/test suite as being not good enough and expressing dissatisfaction with the tool).

We attribute the superior performance of students in the pilot study to their greater experience and a less pressured environment. To address this, future experiments will involve students with comparable experience levels to the overall sample. We hypothesize that incorporating gamification elements aimed at promoting TDD approach to the plugin may yield to even more encouraging results.

Minor issues included students incorrectly selecting a parent directory when opening the project, interrupting the script that committed their progress to the repository. We promptly identified and resolved these problems. General challenges with the committing process stemmed from variations in participants' laptop configurations, which we sometimes deemed unsolvable, opting to collect the final project state at the designated time.

6 CONCLUSIONS

This paper presents the integration and empirical evaluation of INTELLIGAME, a gamification plugin originally designed for Java development, in the realm of JavaScript unit testing. The study aimed to validate INTELLIGAME's effectiveness in JavaScript, adapting it to the popular Jest testing framework. Despite challenges such as transitioning from IntelliJ Community Edition to Ultimate for JavaScript support and addressing differences in coverage information, we successfully ported 19 out of 26 achievements. The controlled experiment with 152 participants revealed mixed perceptions of INTELLIGAME's impact, with achievements influencing testing behaviour and participants' motivation. We will continue with an in-depth analysis of the experimental measures and the code written by the participants to gain more insights and to broaden our knowledge of INTELLIGAME.

REFERENCES

[1] Riccardo Coppola, Tommaso Fulcini, Luca Ardito, Marco Torchiano, and Emil Alègroth. 2023. On Effectiveness and Efficiency of Gamified Exploratory GUI Testing. *IEEE Transactions on Software Engineering* (2023), 1–16. https://doi.org/10.1109/TSE.2023.3348036

[2] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart Nacke. 2011. From Game Design Elements to Gamefulness: Defining "Gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments* (Tampere, Finland) *(MindTrek '11)*. Association for Computing Machinery, New York, NY, USA, 9–15. https://doi.org/10.1145/2181037.2181040

[3] Tommaso Fulcini, Riccardo Coppola, Luca Ardito, and Marco Torchiano. 2023. A Review on Tools, Mechanics, Benefits, and Challenges of Gamified Software Testing. *ACM Comput. Surv.* 55, 14s, Article 310 (jul 2023), 37 pages. https://doi.org/10.1145/3582273

[4] Andreas Jedlitschka and Dietmar Pfahl. 2005. Reporting guidelines for controlled experiments in software engineering. In *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, 10–pp.

[5] José Miguel Rojas and Gordon Fraser. 2016. Code Defenders: A Mutation Testing Game. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 162–167. https://doi.org/10.1109/ICSTW.2016.43

[6] Statista. 2023. Most used programming languages among developers worldwide as of 2023. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ Accessed: 2023-12-05.

[7] Philipp Straubinger and Gordon Fraser. 2023. A Survey on What Developers Think About Testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE).* 80–90. https://doi.org/10.1109/ISSRE59848.2023.00075

[8] Philipp Straubinger and Gordon Fraser. 2024. Improving Testing Behavior by Gamifying IntelliJ. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024.* ACM, 49:1–49:13. https://doi.org/10.1145/3597503.3623339

[9] Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript programs: Challenges and research trends. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–34.

[10] Bo Wang, Aashish Kolluri, Ivica Nikolic, Teodora Baluta, and Prateek Saxena. 2023. User-Customizable Transpilation of Scripting Languages. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 201–229. https://doi.org/10.1145/3586034