2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-7281-9630-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/SANER50967.2021.00062

Automatically Customizing Static Analysis Tools to Coding Rules Really Followed by Developers

Yuki Ueda, Takashi Ishio, Kenichi Matsumoto
Nara Institute of Science and Technology
Nara, Japan
{ueda.yuki.un7, ishio, matumoto}@is.naist.jp

***Abstract*—Automatic Static Analysis Tools (ASATs) detect coding rule violations, including mistakes and bad practices that frequently occur during programming. While ASATs are widely used in both OSS and industry, the developers do not resolve more than 80% of the detected violations. As one of the reasons, most ASATs users do not customize their ASATs to their projects after installation; the ASATs with the default configuration report many rule violations that confuse developers. To reduce the ratio of such uninteresting warning messages, we propose a method to customize ASATs according to the product source code automatically. Our fundamental hypothesis is: A software project has interesting ASAT rules that are consistent over time. Our method takes source code as input and generates an ASAT configuration. In particular, the method enables optional (i.e., disabled by default) rules that detected no violations on the version because developers are likely to follow the rules in future development. Our method also disables violated rules because developers were unlikely to follow them. To evaluate the method, we applied our method to 643 versions of four JavaScript projects. The generated configurations for all four projects increased the ASAT precision. They also increased recall for two projects. The result shows that our method helps developers to focus on their attractive rule violations. Our implementation of the proposed method is available at https://github.com/devreplay/linter-maintainer**

***Index Terms*—Static Analysis Tools, Coding Rule, Coding Convention, Empirical Study**

I. Introduction

Improving the source code readability is critical work in detecting and preventing software defects. Also, maintaining code readability is one of the most costly work. Software projects use the Automatic Static Analysis Tools (**ASATs**) that reveal coding rule violations to reduce maintenance effort automatically [1]. ASATs are used for both open source and industrial projects [2].

Although ASATs are widely used, users ignore more than 80% of violations reported by the tools [3], [4]. UAV visualizes the output of three Java ASATs to help users to understand violation warnings and resolve them more effectively [5]. Additionally, C-3PR automatically fix violations on GitHub [6].

Different from the existing research tools, this study focuses on improving the accuracy of ASAT output. ASAT users ignore violations partly because the detected violations are not attractive to their projects [7]. Although ASATs allow users to customize rules, i.e., they can disable uninteresting rules, most ASAT users use the default rules without customizing them. Improving existing rule selection would help developers to accept ASATs [8].

This paper proposes a method to automatically customize ASAT rules to a software project using its source code. In short, our method takes as input a release version of source code and automatically enables rules that are followed in the current version and disables unfollowed rules. Our method has the following hypotheses.

- If developers are interested in a coding rule, they follow a rule in all of the project source files.
- Developers continue to follow the same set of rules in the software evolution.

The automatically selected rules help developers to keep the same code writing style as the current version.

In this paper, we apply our method to an ASAT named ESLint [9] that detects violations of the ECMAScript coding rules [10]. The contributions of this paper are threefold.

1) We present a technical description of our method to improve ASATs precision using the ASAT violation results (Section III).
2) We investigate the operational status of ASAT usage in four JavaScript projects (Section IV-A).
3) We evaluate our method effectiveness in the four projects (Sections IV-B and IV-C).

II. Motivating Examples

ASATs verify source code according to coding rules for each programming language. The rules are defined to detect well-known, frequent potential bugs and bad practices. As some rules are not always applicable to projects, most ASATs use only widely accepted rules as the default rules. In the case of ESLint [9], the tool checks 56 default rules out of 241 rules. The default rules are selected based on the JavaScript specification ECMAScript 5 [11], which is the standard JavaScript grammar from Ecma International in 2009. As ECMAScript 5 has been supported by legacy browsers such as Internet Explorer, the default configuration helps validate source code that may work on the legacy browsers. Instead, the default rules

978-1-7281-9630-5/21/$31.00 ©2021 IEEE
DOI 10.1109/SANER50967.2021.00062

Listing 1: An example violation of the JavaScript *no-proto* rule that is not included in the default rules of ESLint

```
// An example of violated code
var a = obj.__proto__;
obj.__proto__ = b;

// An example of resolved code
var a = Object.getPrototypeOf(obj);
Object.setPrototypeOf(obj, b);
```

do not include coding rules for ECMAScript 6 and later supported by recent browsers.

Listing 1 shows an example violation of the *no-proto* rule that cannot be detected by the ESLint default rules because the code had been valid until 2008. The "`__proto__`" property has been deprecated as of ECMAScript 3.1, and most browsers have stopped the support to the feature [12]. Developers need to use "`getPrototypeOf`" and "`setPrototypeOf`" methods instead of "`__proto__`". On the other hand, legacy browsers still require "`__proto__`" and do not support the new methods.

Another example to indicate the default rules' deficiency is the *no-emply* rule that disallows empty block statements. Empty block statements may indicate a programming error but may be intentionally used in source code. In case of an intentional empty block, the rule requires to insert the contents such as code comment in the block. However, such a rule violation has no impact on software behavior. As a result, the rule is included in the default rules but often violated by projects.

III. Approach

Our main goal is to improve ASATs' accuracy by reducing false positives and false negatives reported by the ASATs. We decrease false positives by disabling unnecessary rules for a project and decrease false negatives by enabling useful non-default rules.

To achieve the goal, we propose a method to automatically select a set of coding rules followed in a software product's source code. Our method takes as input product source files S and a set of available rules R_{All} implemented in an ASAT. The output is a subset of rules $Config(S)$ that the target project follows. More formally, we consider each coding rule $r \in R_{All}$ as a function $r(S)$ that extracts source code lines violating the rule from source files S. We define the output of our method $Config(S)$ as follows.

$$Config(S) = \{r_i \in R_{All} : \ |r_i(S)| = 0\}$$

The condition $|r_i(S)| = 0$ means that no violations are reported by the rule r_i. For simplicity, we use a single version of the source code for S, while it can be generalized to select rules using multiple versions of source code.

We implemented our method as a tool named *Linter-Maintainer* in a straightforward manner. The tool executes an ASAT of interest to check all rules and then selects non-violated rules. The selected rules are converted into an ASAT configuration file that enables only the non-violated rules. The project can keep its coding style by using the configuration file, specifying necessary and sufficient rules. While we focus on ESLint in this paper, the method applies to various ASATs accepting configuration files to select rules.

Our method has a risk of disabling interesting rules that were accidentally violated in the source code's analyzed version. However, the developers can reduce the risk by manually comparing with generated rules and original rules. Developers also can update rules with every software release using our method. Our tool extracts the current ASAT configuration (i.e., a set of enabled rules) $P(S)$ from a configuration file used in the project to help such a review step. In the case of ESLint, a configuration file can be a JSON, YAML, or JavaScript file whose name starts from `.eslintrc`. If the project does not have a configuration file, $P(S)$ will be the default rule-set. The rules included in both $P(S)$ and $Config(S)$ are true positive rules that are continually followed and enabled rules. The rules included in $P(S)$ but excluded from generated $Config(S)$ are disabled by our method because the violations are not fixed in source code; in other words, the rules report false positives for developers. On the contrary, the rules included in $Config(S)$ but excluded from $P(S)$ are enabled by our method because the rules are followed in source code; violations of the rules in the future versions could be false negatives if the rules are interesting for developers.

IV. Evaluation

To evaluate the effectiveness of our method, we investigate the following research questions:

- **RQ$_1$: What kinds of default rules do projects follow?** As a preliminary analysis, we investigate the number of followed rules and unfollowed rules.
- **RQ$_2$: Are the rules followed by the project consistent over time?** To evaluate our generated rules' consistency, we execute the tool for each release version of the target projects and measure the accuracy of rule selection.
- **RQ$_3$: Do the generated rules improve accuracy of ASAT?** To evaluate future violation detection effectiveness, we measure the accuracy of coding rule violations detected by the tool configured with our method.

In this study, we focus on JavaScript that is one of the most popular languages. As the ASAT, we use ESLint that is widely used in JavaScript projects [13]. To use consistent rules across the projects and their source code history, we use ESLint version 7.4.0, although the projects could not use the version before its release.

We investigate four JavaScript projects that explicitly configure ESLint. The projects are included in BugsJS [14] that selected ten server-side application projects using the

TABLE I: The number of release versions for each project

Project	Major	Minor	Maintenance	All
bower	1	19	89	96
eslint	7	119	165	223
hexo	4	30	120	126
karma	5	29	187	198
Total	17	197	561	643

TABLE II: RQ$_1$: Followed rules distribution

Category	# of rules	# of default rules
Followed	64	38
Unfollowed	95	4
Specific	82	15
Total	241	57

GitHub stargazers count. From those projects, we select projects, including ESLint configuration files.

The four projects have 643 package versions. We selected 561 versions whose version numbers follow the “x.y.z” format so that we can classify them into major, minor, and maintenance releases. We excluded 82 extra versions such as ”x.y.z-b” and “test-version” because the release granularity is unclear. Table I shows the number of release versions for each project. The major releases are recognized by their version numbers “x.0.0”. The minor releases are recognized by “x.y.0” including major release versions. The maintenance releases include all the versions.

A. RQ$_1$ (preliminary study): What kinds of default rules do projects follow?

To answer the question, we extracted $Config(S)$ from the latest release for each project and compare the rules with the default rule-set of the ESLint. We classified all of the rules into three categories.

- **Followed**: The rules are followed by all of the four projects. In other words, they are included in $Config(S)$ of four projects.
- **Unfollowed**: The rules are violated in all the projects. They are never included in $Config(S)$.
- **Specific**: The rules are followed in some projects but unfollowed in some projects.

Table II shows the number of rules in each category. It shows that the default rule-set is not always followed in the projects. We analyzed the details for each category.

Followed rules: All the projects follow the coding rules detecting potential errors. Only 38 out of 64 followed rules (59.4%) are included in the default rule-set. The projects also follow additional rules including *no-proto*, *prefer-const*, and *dot-notation*. We explained the *no-proto* rule in Section II. All of the developers follow those rules probably because the rule violations will change the program behavior.

Unfollowed rules: All the target projects do not follow 95 rules. We found that 4 out of the 95 rules are included in the default rules. The four rules are *no-empty*, *no-undef*, *no-unused-vars*, and *no-extra-semi*. We explained the *no-empty* rule in Section II. Unlike other default rules, these four rule violations may reduce code readability but they do not affect the program behavior.

Listing 2: Examples of JavaScript *no-unreachable* and *no-constant-condition* rule violations

```
// Example of no-unreachable violation
function foo() {
    if (Math.random() < 0.5) {
        return True;
    } else {
        throw new Error();
    }
    // Following line is un-reachable
    return False;
}

// Example of no-constant-conditon violation
if (false) {
    // Following line can not be executed
    doSomethingUnfinished();
}
```

Specific rules: We found that 82 rules are not consistent across the projects, and 15 of the 57 default rules are here. For example, *no-unreachable*, *no-constant-condition*, and *no-cond-assign* rules are included. Listing 2 shows examples of *no-unreachable* and *no-constant-condition* violations. The *no-unreachable* rule disallows source code that cannot be executed due to the `return`, `throw`, `break`, and `continue` statements. Also, the *no-constant-condition* rule disallows a constant expression in a conditional expression. Unlike rule violations causing a runtime error such as *no-proto*, that source code is unexecuted. If developers are intentionally writing such source code, the rules report false positives for the developers.

B. RQ$_2$: Are the rules followed by the project consistent over time?

We evaluate the accuracy of rule selection for each of release versions. A software project has a sequence of release versions $S_1, S_2, \cdots, S_n$. We regard a rule $r \in Config(S_i)$ for an intermediate version S_i $(1 \leq i < n)$ as correct if $r \in Config(S_n)$, i.e. the rule is followed in the latest release. A rule $r \in Config(S_i)$ is regarded as false positive if $r \notin Config(S_n)$. A rule $r \in (R_{All} \setminus Config(S_i))$ is false negative if $r \in Config(S_n)$.

We used three release sequences: major releases, minor releases, and maintenance releases. For each version S_i in a sequence, we extracted $Config(S_i)$ and calculated its precision, recall, and F-value. The baseline is the accuracy of $P(S_n)$, that is a set of project-specific rules defined in the configuration file in the latest release.

Table III shows the mean and median precision, recall, and F-value of the rules selected by our method for each intermediate version. The result shows that our method can accurately select coding rules. In the target projects, rules selected by the maintenance releases are the most accurate. On the other hand, the rules selected by the

TABLE III: Accuracy of rule selection in RQ_2

Project	Precision		Recall		F-value	
	mean	med	mean	med	mean	med
bower (n=96)						
Major	0.93	0.94	0.95	0.95	0.94	0.95
Minor	0.99	0.99	0.98	0.99	0.98	0.99
Maintenance	0.99	1.00	0.99	1.00	0.99	1.00
Project	0.90	0.89	0.34	0.34	0.50	0.50
eslint (n=223)						
Major	0.98	0.99	0.90	0.99	0.93	0.98
Minor	0.99	1.00	0.98	1.00	0.98	1.00
Maintenance	0.99	1.00	0.99	1.00	0.99	1.00
Project	0.29	0.00	0.16	0.00	0.21	0.00
hexo (n=126)						
Major	0.92	0.92	0.92	0.93	0.92	0.93
Minor	0.98	0.99	0.98	1.00	0.98	1.00
Maintenance	0.99	1.00	0.99	1.00	0.99	1.00
Project	0.71	0.89	0.27	0.33	0.39	0.48
karma (n=198)						
Major	0.77	0.70	0.90	0.88	0.82	0.78
Minor	0.97	0.99	0.96	1.00	0.96	0.99
Maintenance	0.99	1.00	0.99	1.00	0.99	1.00
Project	0.83	0.81	0.39	0.39	0.53	0.53

major versions are slightly inaccurate. In other words, the coding rules of the projects slightly changed over time. Developers should update the ASAT configuration file periodically.

The project configuration recall is the worst, and the precision is also lower than our method. It means that developers did not recognize the rules that they follow. Our method lets developers know such potential rules.

C. RQ_3: Do the generated rules improve accuracy of ASAT?

We compare the accuracy of violations detected by configuration files generated by our method and manually defined by the project. To approximate the number of violations resolved by developers in the evolution process, we classify rule violations to *Resolved* and *Unresolved*. We regard violations of a rule r in a file version f_i as *Resolved* if there exists a clean version f_k $(i < k)$ such that $|r(f_k)| = 0$. For each clean file version, only the maximum number of violations detected between the previous clean version and the current version is counted as *Resolved*. Figure 1 shows an example history of the numbers of violations of a rule r in a file f. The versions $n-5$ through $n-3$ violated the rule, but the version $n-2$ did not violate the rule. In this case, we count the maximum number of violations in the span, i.e. 10 as *Resolved* violations. Although developers might have resolved some other violations between the version $n-5$ and $n-4$, we do not count them because some violations such as *max-params* may reappear in the successive versions. We conservatively assume that the clean version $n-2$ is the time of resolution. If our method enabled the rule r in the span, we count them as true positives because they are detected. If our method disabled the rule r in the span, we count them as false negatives. In Figure 1, five violations in the version $n-1$ are counted as *Unresolved* because some of them still exist in the latest version n. If our method enabled the rule r in the span, we count them as false positives. Otherwise, they are true negatives because they are undetected and unresolved.

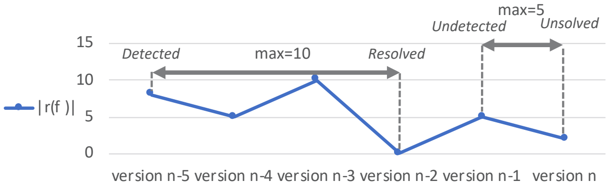

Fig. 1: Example of counting rule violations for RQ_3

To simulate the effect of our method, we suppose three use case scenarios: Developers update the configuration $Config(S_i)$ in every major/minor/maintenance version. We apply the generated configurations to detect violations in every maintenance release. We classify *Resolved* and *Unresolved* violatoins to true positives, false positives, false negatives, and true negatives as described above using $Config(S_i)$ and the history of detected violations.

Table IV shows the resultant precision, recall, and F-value of five settings. The Major, Minor, and Maintenance rows show the results of configuration files generated by our method. The Project row shows the result of configuration files $P(S_i)$ included in the projects. The Default row shows the result of the ESLint default configuration.

The best result of our method is the case that configuration files are updated every major release. Frequent updates might accidentally disable some interesting rules due to temporary violations. In all the projects, the generated rule configurations have higher precision than the manually selected project configurations. This is because our method disabled uninteresting rules for developers. Our method also increased recall in two projects, eslint, and karma. This might be because developers in the projects resolved many violations without the support of the ASAT. Our method could enable the rules to detect future violations.

V. Threath to validity

A. Internal validity

Our hypotheses compromise two cases. First, our method enables the rules that have no impact on project code even if the rules are enabled or disabled. For example, *no-proto* rule will get enabled even though "`__proto__`" is not used in the code.

To minimize false positives, our method disables a rule if one file does not comply with the rule, even if the other files in the project adhere to the rule. Although we could use a relaxed condition, e.g., $|r_i(S)| \leq k$ to select rules, we avoid introducing such a parameter because giving an appropriate parameter is difficult for developers.

TABLE IV: Accuracy of detected violations in RQ_3 (n=405,094,892)

Category	bower (n=4,174,637)			eslint (n=363,074,539)			hexo (n=14,416,765)			karma (n=23,428,951)		
	Precision	Recall	F-val	Precision	Recall	F-val	Precision	Recall	F-val	Precision	Recall	F-val
Major	0.95	0.02	0.05	0.27	0.03	0.06	0.56	0.03	0.06	0.87	0.19	0.31
Minor	0.97	0.01	0.01	0.37	0.02	0.04	0.46	0.02	0.03	0.84	0.05	0.10
Maintenance	0.95	0.01	0.01	0.44	0.02	0.04	0.46	0.02	0.03	0.83	0.05	0.09
Project	0.80	0.04	0.07	0.16	0.00	0.00	0.51	0.04	0.07	0.55	0.01	0.02
Default	0.80	0.04	0.07	0.34	0.01	0.02	0.51	0.04	0.07	0.55	0.01	0.02

To keep rule consistency, we execute the ASATs using ESLint as of October 2020 through all versions. We evaluated the rule-set's accuracy by counting the number of lines where the rule violation occurred and was resolved.

Even if the file is renamed and deleted, we counted them as resolved violations. This may overestimate the number of resolved violations and underestimate the number of false positives. However, the analysis does not affect the total number of rule violations in the projects.

B. External validity

ESLint supports third-party plug-ins introducing additional rules. We did not evaluate such rules to equalize the number of rules through the projects. Additionally, our evaluation only included JavaScript projects using ESLint. The result may be dependent on the programming language characteristics.

VI. Summary and future works

This paper proposed a method to automatically customize the ASAT coding rules to software projects' source code. Our method enables rules followed in the source code and disabled rules violated in the source code. We implemented our method to generate an ESLint configuration file and applied it to 643 versions of four JavaScript projects. The result shows that our method selected necessary and sufficient rules for the projects. Our method increased precision for all the projects and increased recall for two projects. Our method is promising to help developers automatically configure their ASATs.

In future work, we have three research extensions. First, we will extend the evaluation to other ASATs for different programming languages, such as PMD for Java and Pylint for Python. Secondly, we would like to improve the recall of ASATs by combining our method with violation prioritization approaches [15], [16]. Finally, to evaluate the real development usefulness, we plan to the field survey for the developers.

VII. Acknowledgement

This work was supported by JSPS KAKENHI Grant Numbers JP18H03221, JP18KT0013, JP20H05706 and JP20J15163.

References

[1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, 2013, pp. 672–681.

[2] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, p. 58–66, 2018.

[3] F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu *et al.*, "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, 2012, pp. 50–59.

[4] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of SonarQube," in *Proceedings of the 27th International Conference on Program Comprehension (ICPC'19)*, 2019, pp. 209–219.

[5] T. Buckers, C. Cao, M. Doesburg, B. Gong, S. Wang, M. Beller, and A. Zaidman, "UAV: Warnings from multiple automated static analysis tools at a glance," in *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*, 2017, pp. 472–476.

[6] A. Carvalho, W. Luz, D. Marcílio, R. Bonifácio, G. Pinto, and E. D. Canedo, "C-3PR: A bot for fixing static analysis violations via pull requests," in *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*, 2020, pp. 161–171.

[7] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, vol. 1, 2016, pp. 470–481.

[8] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*, 2018, pp. 38–49.

[9] N. C. Zakas, B. Mills, T. Nagashima, and M. Djermanovic, "ESLint - Pluggable JavaScript linter," https://eslint.org/, 2020.

[10] Ecma International, "Standard ECMA-262," https://www.ecma-international.org/publications/standards/Ecma-262.htm, 2020.

[11] ——, "ECMAScript Language Specification, 5.1 edition," https://www.ecma-international.org/ecma-262/5.1/, 2016.

[12] ESLint User guide, "Disallow Use of `__proto__` (no-proto)," https://eslint.org/docs/rules/no-proto, 2020.

[13] D. Kavaler, A. Trockman, B. Vasilescu, and V. Filkov, "Tool choice matters: JavaScript quality assurance tools and usage outcomes in GitHub projects," in *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*, 2019, pp. 476–487.

[14] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, "Bugsjs: a benchmark and taxonomy of javascript bugs," *Software Testing, Verification and Reliability*, 2019.

[15] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *International Static Analysis Symposium.* Springer, 2003, pp. 295–315.

[16] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'08)*, 2008, pp. 41–50.