Automating Linter Configurations for Coding Standards via DSL-Driven, LLM-Based Compilation

Appendix A
Coding Standards and Tools in Practice

1) Motivation: We are first interested in understanding the trends of coding standards and their corresponding linter tools in real-world software development.

2) Approach: We investigate trends in coding standards and tools in software development on GitHub. Building on prior research and developer practices [1]–[8], we use keyword-based search method to collect projects related to coding standards and their corresponding tools. To ensure for project quality, we consider only projects with at least five stars. For coding standard-related projects, we search using keywords such as coding standard, code standard, convention, and style guide, then manually verify their relevance, identifying 1,066 projects. For tool-related projects, we use keywords like coding standard tool, code standard tool, convention tool, style guide tool, and linter, followed by manual verification, yielding 1,692 projects.

To examine the long-term popularity of coding standards and tools, we group the projects by their creation year. To determine whether they are actively maintained, we check if projects have commits within the current year. Commits serve as an important indicator of project maintenance, reflecting their continued development and engagement.

3) Result: Figure 1 shows the number of coding standards and tool projects from 2009 to 2024. By 2024, there are 1,066 coding standard projects, with new projects created annually ranging from 2 ~154. The number of new projects gradually increased from 2009 but slowed after 2016. Despite this, 20% (218 out of 1,066) of projects are still actively updated in 2024. Typically, coding standards remain stable as they are tied to specific organizational or project requirements, but the fact that 20% are still maintained suggests they continue to evolve in response to changes in development practices and technology.

For tool projects related to coding standards, by 2024, there are 1,692 total projects, with new projects ranging from 4 ~202 annually. Similar to coding standards, new tool projects gradually grew until 2015, fluctuated between 2016 and 2020, and began to slow down after 2021. Despite this, 47% of tool projects (787 out of 1,692) remain actively maintained in 2024. Unlike coding standards, tools for coding standards are more dynamic, evolving rapidly with new features, technologies, and practices.

Given the large number of coding standards and tools, along with their frequent updates, developers must continuously track changes, making configuration generation an ongoing effort rather than a one-time task.

Summary: Projects related to coding standards and their corresponding tools on GitHub are continuously increasing and being updated.

Appendix B
DSL for Coding Standards and Linter Configurations

A. DSL for Coding Standards and Tool Configurations

1) Motivating Examples for DSL Design: Design a DSL that represents coding rules in a tool-agnostic, structured, precise and readble way is essential for reliable compilation from NL coding standards to linter configurations. While both NL coding standards and machine-readable linter configurations can represent coding rules, they do not meet the requirements.

NL is tool-agnostic and readable but lacks structure and precision. A NL sentence of coding standards may or may not contain rules and can embed multiple layers of information, such as rule types, objects being checked, and constraint types. For instance, in Figure 2, the sentences in the *NL box* illustrate this issue. In the Google Java coding standard, sentence i does not have rules, while sentence j has two rules. The two rules have mixed information: (1) "should always" and "must" indicate mandatory rules; (2) "block tag" and "these tags" refer to the four types of block tags being checked; (3) "appear in this order" and "not have" imply order and negation constraint relationships.

In contrast, machine-readable formats, such as XML, which are structured and precise, but tool-dependent and often too abstract to convey complete meanings. In the *Linter Configuration Format box* of Figure 2, the Checkstyle configuration for sentence j consists of configuration names, option names and values. However, the configuration names, option names, and values are abstract and incomplete, not enough to convey the exact meaning. Moreover, linter configurations such as module names and formats are specific to Checkstyle and cannot be reused by other linters, which often use different settings and formats.

To provide a tool-agnostic, structured, precise and readable representation, we design the DSL shown in the *domain-specific language* box from Figure 2. The corresponding DSL representations of sentence j are "*Mandatory: Order of [BlockTag] is [@param, @return, @throws, @deprecated]*" and "*Mandatory: No [EmptyDescription] for [@param, @return, @throws, @deprecated]*". The DSL independently expresses each rule, clearly separating rule types (e.g., *"Mandatory:"*), objects (enclosed in *"[]"*), and constraint relationships (e.g., *"Order of"* and *"No"*).

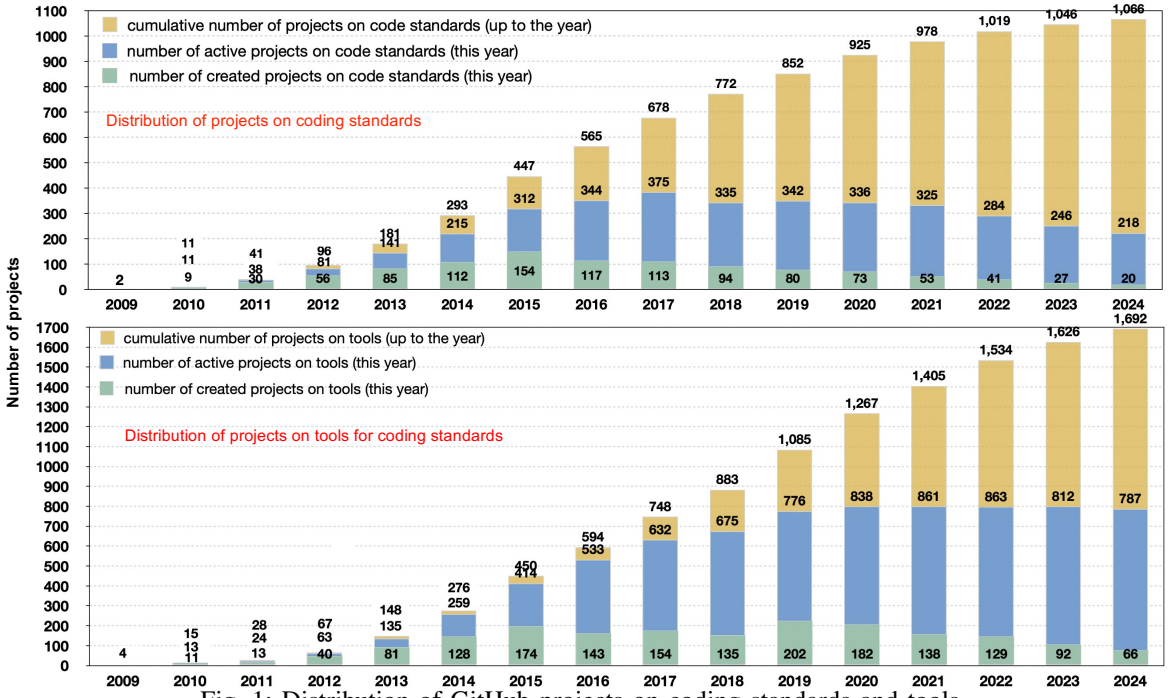

Fig. 1: Distribution of GitHub projects on coding standards and tools

Natural Language (NL)

Google Java Style Guide:

- **Sentence i:** Sometimes there is more than one reasonable way to convert an English phrase into camel case.
- **Sentence j:** The standard block tags @param, @return, @throws, @deprecated should always appear in this order, and these tags must not have an empty description.

Explanation:

- **Sentence i:** it does not contain rules.
- **Sentence j:** it contains rules but **mixes different level information** such as the number of rules, rule types, objects being checked by rules, and corresponding constraint types of rules.
  - the sentence contains **two rules**;
  - "should always" and "must" indicate that the two rules are **mandatory rules**;
  - "block tag" and "these tags" refer to the **four types of block tags being checked**;
  - "appear in this order" and "not have" imply **constraint relationships**, such as order and negation.

NL Summary:

Good: general; readable **Bad:** too free → lacking structure and precision

Linter Configuration Format (e.g., XML)

Checkstyle Configuration:

```
<module name='AtclauseOrder'>
  <property name="tagOrder" value="@param, @return, @throws,
   @deprecated"/> </module>
<module name='NonEmptyAtclauseDescription'>
  <property name="tokens" value="PARAM_LITERAL,
   RETURN_LITERAL, THROWS_LITERAL,
   DEPRECATED_LITERAL"/> </module>
```

Explanation:

- The module names, property name and values are **too vague and abstract**, making it difficult to understand their specific meanings.
- The **format** and **configuration settings**, such as module names, are **specific to** Checkstyle, while other linters use distinct formats and settings.

Linter Configuration Format Summary:

Good: structured; precise **Bad:** tool-dependent; hard to read

Domain-Specific Language (DSL)

DSL Representation:

Mandatory: Order of [BlockTag] is [@param, @return, @throws, @deprecated] ;
Mandatory: No [EmptyDescription] for [@param,@return,@throws,@deprecated]

Explanation:

- **Each rule** is represented **independently**.
- The prefix "Mandatory:" denotes the **rule type**, indicating it must be satisfied.
- "Order " and "No" denote different **constraint relationships**: order and negation.
- **Checking objects** are enclosed in [], distinguishing them from the **constraint relationship**.

DSL Summary:

Good: general; readable; structured; precise

The DSL We Designed:

```
RuleSet    ::= Rule [';' Rule]*
Rule       ::= ['Optional:' | 'Mandatory:'] Constraint [ExceptRule]*
Constraint ::= TermList [Operator TermList]*
             | 'No' Constraint
             | 'Order of' TermList ['is' | 'is not'] TermList
             | 'Number of' TermList Operator TermList
             | 'if' Constraint 'then' Constraint
ExceptRule ::= 'Except' Constraint
Operator   ::= [.]+  (e.g., 'is' | 'is not' | 'have' | 'after' |  '=' | 'Add ' | '...' )
TermList   ::= Term [', ' Term]*
Modifier   ::= Word (e.g., 'some' | 'each' | 'all' | 'first' | 'last' | '...')
Term       ::= '[' PLterm ']'
             | Modifier* PLterm
             | PLterm 'of' PLterm
Word       ::= [a-zA-Z]+
PLterm     ::= [.]+
```

Note: PLterm *denotes terminology in programming languages like Java;*
'. ' means to match any single character except for newline.
'...' indicates that more words can be included if needed;
'' means zero or more repetitions; '+' means one or more repetitions.*

Fig. 2: Comparison of coding rule representations (NL, linter configuration, and DSL)

2) Information Source for Designing DSL: To design the DSL, we first select research objects of coding standards and linters. Java, with its mature ecosystem and rigorous coding practices [9], represents a language with long-standing conventions, making it an ideal choice. Given Google's coding standards' high recognition across software projects and the industry [10], we choose the Google Java style guide. To enforce these standards, we select Checkstyle [11] for Java coding standards. Checkstyle is highly configurable, widely used, and have been studied extensively [8], [12]–[14]. To understand linter configurations, we refer to Checkstyle documentation [15], which provides detailed information about linter configurations.

3) Process of Designing DSL: We use a card sorting approach [16] to define DSL by analyzing a sample of 320 sentences with a confidence level of 95% and a confidence interval of 5 drawn from all sentences from both Google JavaScript style guide and Checkstyle documentation. The process of designing DSL has two iterations and we used two evaluators. In the first iteration, we randomly sample 175 sentences with a confidence level of 95% and a confidence interval of 5 from 320 sentences. Two authors first independently represent each sentence using their designed DSLs, and then they discuss to construct a DSL. In the second iteration, two of the authors independently represent remaining sentences with the DSL. If the DSL is not enough to represent the rule of a sentence, they annotate the sentence with a brief description. They find that there are no sentences that cannot be represented using the DSL. We use Cohen's Kappa measure [17] to examine the agreement between the two authors. The Kappa value is 0.7, which indicates a high agreement between two authors. Finally, two authors discuss the disagreements to reach an agreement.

4) DSL for Coding Standards and Linter Configurations: Figure 2 shows the designed DSL. The detailed explanation is as follows:

- *RuleSet* represents sets of *Rules*, where individual rules are separated by semicolons.
- A *Rule* comprises a *RuleType*, a *Constraint*, and optionally any number of *ExceptRule*. The *RuleType* can be 'Mandatory' or 'Optional', indicating whether the rule must be followed or is optional. The *ExceptRule* specifies exceptions to a *Constraint*. The *Constraint* has five types:

(1) *TermList [Operator TermList]**: Denotes the relationship that TermList must satisfy, expressed via operators. For example, "Checks for braces around code blocks" from the Checkstyle documentation is represented in the DSL as "*Mandatory: [CodeBlock] have [Brace]*".

(2) *'No' Constraint*: The *Constraint* is prohibited. For example, in the Figure 2, the NL description of the Google Java coding standard, "these tags must not have an empty description" is represented in the DSL as "*Mandatory: No [EmptyDescription] for [@param, @return, @throws, @deprecated]*".

(3) *'Order of' TermList ['is' | 'is not'] TermList*: Defines the ordering requirements where one TermList should or should not follow another TermList. For example, in the Figure 2, the NL description of the Google Java coding standard, "the standard block tags @param, @return, @throws, @deprecated should always appear in this order" is represented in the DSL as *"Mandatory: Order of [BlockTag] is [@param, @return, @throws, @deprecated]"*.

(4) *'Number of' TermList Operator TermList*: Specifies constraints on the quantity of *TermList*. For example, the Google Java coding standard "each annotation is listed on a line of its own (that is, one annotation per line)" is represented in DSL as: *"Mandatory: Number of [Annotation] = 1 for each [Line]"*.

(5) *'if' Constraint 'then' Constraint*: Represents a constraint should be satisfied when the another constraint satisfied. For example, the Checkstyle documentation, "Allow to ignore enums when left curly brace policy is EOL" is represented in DSL as: *"Optional: if [LeftCurlyPolicy] is [EOL] then [Enum] of [LeftCurly] is not [EOL]"*.

- *Operator* denotes the relationship *TermList*s should satisfy.
- *TermList* is a list of *Term*s separated by commas.
- *PLterm* represents terminology specific to programming languages, such as "BlockTag" and "@param" in the DSL representation of the Figure 2. It can be any sequence of characters, excluding newlines.

REFERENCES

[1] B. Goncharenko and V. Zaytsev, "Language design and implementation for the domain of coding conventions," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, 2016, pp. 90–104.

[2] N. Ogura, S. Matsumoto, H. Hata, and S. Kusumoto, "Bring your own coding style," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 527–531.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, 2014, pp. 281–293.

[4] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia, "Code convention adherence in evolving software," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 504–507.

[5] A. J. Simmons, S. Barnett, J. Rivera-Villicana, A. Bajaj, and R. Vasa, "A large-scale comparative analysis of coding standard conformance in open-source data science projects," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[6] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *2008 IEEE International conference on software maintenance*. IEEE, 2008, pp. 277–286.

[7] (2024) Eslint. [Online]. Available: https://github.com/eslint/eslint

[8] C. C. Yiu, "Checkstyle for legacy applications [j]," *Itestra De*, 2023.

[9] M. M. Abdallah and M. M. Al-Rifaee, "Java standards: A comparative study," *International Journal of Computer Science and Software Engineering*, vol. 6, no. 6, p. 146, 2017.

[10] (2024) Google style guide. [Online]. Available: https://github.com/google/styleguide

[11] (2024) Checkstyle. [Online]. Available: https://github.com/checkstyle/checkstyle

[12] E. Torunski, M. O. Shafiq, and A. Whitehead, "Code style analytics for the automatic setting of formatting rules in ides: A solution to the tabs vs. spaces debate," in *2017 Twelfth International Conference on Digital Information Management (ICDIM)*. IEEE, 2017, pp. 6–14.

[13] B. Loriot, F. Madeiral, and M. Monperrus, "Styler: learning formatting conventions to repair checkstyle violations," *Empirical Software Engineering*, vol. 27, no. 6, 2022.

[14] S. H. Edwards, N. Kandru, and M. B. Rajagopal, "Investigating static analysis errors in student java programs," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, 2017, pp. 65–73.
[15] (2024) Offical documentation of checkstyle. [Online]. Available: https://checkstyle.sourceforge.io/checks.html
[16] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.
[17] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica: Biochemia medica*, 2012.