



# Styler: learning formatting conventions to repair Checkstyle violations

Benjamin Lorient<sup>1</sup> · Fernanda Madeiral<sup>2</sup> · Martin Monperrus<sup>2</sup>

Accepted: 24 September 2021 / Published online: 6 August 2022

© The Author(s) 2022

## Abstract

Ensuring the consistent usage of formatting conventions is an important aspect of modern software quality assurance. To do so, the source code of a project should be checked against the formatting conventions (or rules) adopted by its development team, and then the detected violations should be repaired if any. While the former task can be automatically done by format checkers implemented in linters, there is no satisfactory solution for the latter. Manually fixing formatting convention violations is a waste of developer time and code formatters do not take into account the conventions adopted and configured by developers for the used linter. In this paper, we present STYLER, a tool dedicated to fixing formatting rule violations raised by format checkers using a machine learning approach. For a given project, STYLER first generates training data by injecting violations of the project-specific rules in violation-free source code files. Then, it learns fixes by feeding long short-term memory neural networks with the training data encoded into token sequences. Finally, it predicts fixes for real formatting violations with the trained models. Currently, STYLER supports a single checker, Checkstyle, which is a highly configurable and popular format checker for Java. In an empirical evaluation, STYLER repaired 41% of 26,791 Checkstyle violations mined from 104 GitHub projects. Moreover, we compared STYLER with the IntelliJ plugin CHECKSTYLE-IDEA and the machine-learning-based code formatters NATURALIZE and CODEBUFF. We found out that STYLER fixes violations of a diverse set of Checkstyle rules (24/25 rules), generates smaller repairs in comparison to the other systems, and predicts repairs in seconds once trained on a project. Through a manual analysis, we identified cases in which STYLER does not succeed to generate correct repairs, which can guide further improvements in STYLER. Finally, the results suggest that STYLER can be useful to help developers repair Checkstyle formatting violations.

**Keywords** Coding conventions · Linter · Format checker · Checkstyle · Formatting violations · Automatic repair

---

Communicated by: Andrea De Lucia

✉ Fernanda Madeiral  
fer.madeiral@gmail.com

Extended author information available on the last page of the article.

## 1 Introduction

Coding conventions are widely recognized as a means to improve the internal quality of software systems (Prause and Jarke 2015). They are rules that developers agree on for writing code, which encode best coding practices, widely adopted standards, or developers' preferences. The usage of coding conventions helps to reduce style deviations, which are nothing but distracting noise when reading code (Spinellis 2011; Prause and Jarke 2015).

However, keeping all source code files of a project compliant with the coding conventions agreed by a development team is a challenge. For that, two main activities must be performed: the *detection* and the *repair* of coding convention *violations*. The detection of coding convention violations can be automatically performed using *linters*. A linter is a tool that statically analyzes code to check its compliance with rules and warns software developers when rule violations are found. The usage of linters also brings challenges because the developers need to create a configuration according to their adopted conventions so that the linter detects the right violations (not more and not less). Nevertheless, in this paper, we focus on the latter task, i.e., the repair of violations, which is a little researched problem.

To repair coding convention violations, developers can either perform fixes manually or use automated solutions that produce fixes. Manually fixing these violations is a waste of valuable developer time. Considering *formatting convention violations*, which are the focus of this paper, developers could use *code formatters* as automated solution. However, this alternative is also not satisfactory. With code formatters, the key problem is that they do not take into account the project-specific rules, those that are configured by developers for the used linter.

Inspired by the problem statement of program repair (Monperrus 2018), we state in this paper the problem of automatically repairing formatting violations: *given a program, a set of format checker rules, and one rule violation, the goal is to modify the source code formatting so that no violation is raised by the format checker. A format checker is a linter, or a part of a linter, that focuses on formatting checks, since linters cover several classes of coding conventions, e.g., naming and formatting.*

In this paper, we explore that problem in the context of Checkstyle<sup>1</sup>, a popular format checker for the Java language. We present STYLER, a tool dedicated to fixing formatting violations in Java source code. The uniqueness of STYLER is its applicability to any formatting convention because its approach is not based on specific format checker rules. The key idea behind STYLER is the usage of machine learning to learn the formatting conventions that are used in a software project. The learning is based on training data generated by STYLER through the modification of source code files to trigger violations of the formatting rules configured by developers for a given project. Once trained, STYLER predicts changes on formatting characters (e.g., whitespaces) to fix formatting convention violations happening in the wild. Technically, STYLER encodes Java source code containing formatting violations into abstract token sequences and uses sequence-to-sequence machine learning models based on long short-term memory neural networks (LSTMs).

To evaluate STYLER, we conducted a large scale experiment using 26,791 Checkstyle formatting violations mined from 104 GitHub projects. Based on our research questions, we found out that STYLER repairs many violations (41%) from a diverse set of formatting rules (24/25). It generally performs better for fixing violations related to horizontal whitespace between Java tokens than violations related to tabulations and line length.

<sup>1</sup><https://checkstyle.sourceforge.io/>, last access: 2020-07-13

Moreover, STYLER produces smaller repairs compared to the state-of-the-art machine learning formatters (Allamanis et al. 2014; Parr and Vinju 2016) and the IntelliJ plugin CHECKSTYLE-IDEA (CheckStyle-IDEA 2021). Finally, STYLER repairs violations in seconds, once it is trained for a given project.

To sum up, our contributions are:

- A novel approach to fix violations of code formatting conventions, based on machine learning. The approach is able to learn project-specific formatting rules with a self-training data generation strategy and repair formatting rule violations with a sequence-to-sequence machine learning model;
- A tool, called STYLER, which implements our approach in the context of Java and Checkstyle, to repair Checkstyle formatting violations. The tool is made publicly available<sup>2</sup> for future research and usage;
- A dataset of real-world Checkstyle violations mined from GitHub repositories. The dataset is publicly available<sup>3</sup> for future research;
- A comparative experiment of the performance of STYLER against the state-of-the-art code formatters (CheckStyle-IDEA 2021; Allamanis et al. 2014; Parr and Vinju 2016). The results of the experiment are also publicly available<sup>4</sup> for the sake of open science.

The remainder of this paper is organized as follows. Section 2 presents the background of this work. Section 3 presents STYLER in detail, including its workflow and technical principles. Section 4 presents the design of our experiment for evaluating STYLER and comparing it with three code formatters. The experimental results are then presented in Section 5. Section 6 presents discussions, and Section 7 presents the related works. Finally, Section 8 presents the concluding remarks of this work.

## 2 Background

Coding conventions play an important role in software development and maintenance. In this section, we present a background on coding conventions and tools that help developers enforce them. In addition, we report on a study of the usage of Checkstyle, a tool that statically checks Java code against a specified set of coding conventions.

### 2.1 Coding conventions

*Coding conventions*, also known as coding style and coding standards, are *rules* that developers agree on for writing code. The usage of coding conventions does not affect the behavior of software systems. Instead, developers use them to improve code readability and maintainability. Although not all coding practices are perceived by developers as enhancing code readability (Santos and Gerosa 2018), they help to reduce style deviations, which are nothing but distracting noise when reading code (Spinellis 2011; Prause and Jarke 2015).

There are several kinds of coding conventions, e.g., conventions related to naming and formatting. In this paper, we focus on the latter, i.e., *formatting* conventions. Formatting

---

<sup>2</sup><https://github.com/KTH/styler/>

<sup>3</sup><https://github.com/KTH/checkstylelr/>

<sup>4</sup><https://kth.github.io/styler-experiments/>

refers to the appearance or the presentation of the source code. One can change the formatting by using, for instance, *non-printable characters*, such as spaces, tabulations, and line breaks. In free-format languages such as Java and C++, the code formatting does not change the abstract syntax tree of programs. In non-free-format languages, such as Python and Haskell, the formatting is even related to behavior, which means that correcting formatting issues can fix bugs.

To exemplify formatting conventions, consider Fig. 1, which shows two well-known ways that developers may follow when placing left curly braces in code blocks. Note that one way is to place the left curly brace in a new line (Fig. 1a) while another way is to place it at the end of the conditional expression line (Fig. 1b). The way to actually do it in a software project depends on what the project's development team chooses. Agreeing on coding conventions to be followed in a software project is important to avoid edit wars and endless debates.

## 2.2 Detection of coding convention violations

A challenge faced by developers is to keep their code compliant with the agreed coding conventions. Basically, every new change in the code must satisfy the adopted coding conventions. Manual analysis of code changes for checking if they do not violate the adopted coding conventions is time-consuming and error-prone. To do so automatically, one can use *linters*. A linter is a tool that statically analyzes code to check its compliance with rules and warns software developers when rule violations are found. The rules might be related to functional problems, such as resource leakage or incorrect logic, and maintainability problems, such as non-compliance with best practices or violations of style conventions (Beller et al. 2016). As a side note, the literature does not consistently relate linters and automated static analysis tools (abbreviated as ASATs, also known as static analyzers). However, we understand that an automated static analysis tool is any tool that analyzes source code without the need to run it, including, for instance, tools for software analytics. Therefore, in this paper, we consider that any linter belongs to the family of automated static analysis tools, but that automated static analysis tools are not all about analyzing code against a set of rules.

Linters can be usually integrated into IDEs and build tools. On the one hand, when integrated into IDEs, developers may manually run the linter before they commit their changes. If they do not do it, they might face a lot of violations raised by the linter after the end of the building step for a release or for shipping the program. On the other hand, when a linter is integrated into build tools, it might be automatically executed in Continuous Integration (CI) environments. The important coding conventions might be configured to make CI builds break when they are violated. This way, developers are forced to repair coding convention violations early in the software development process.

<pre> if (condition) {     // do something } </pre>	<pre> if (condition) {     // do something } </pre>
-----------------------------------------------------	-----------------------------------------------------

(a) Left curly in a new line.

(b) Left curly at the end of line.

**Fig. 1** Two conventions for placing a left curly brace

Several linters have been developed for different programming languages. Some examples include ESLint<sup>5</sup> for JavaScript, Pylint<sup>6</sup> for Python, StyleCop<sup>7</sup> for C#, and RuboCop<sup>8</sup> for Ruby. For Java, which is our target language in this paper, a commonly used linter is Checkstyle<sup>9</sup>. Checkstyle is composed of several *checks* that encode style-related rules. For instance, Checkstyle contains a check named *LeftCurly* that checks for the placement of left curly braces for code blocks, which is about the example illustrated in Fig. 1. Developers specify in a configuration file, then, their coding conventions by selecting and configuring Checkstyle checks. We refer to this configuration file as *Checkstyle ruleset* and, hereafter, we refer to Checkstyle checks as Checkstyle rules. Finally, Checkstyle is a flexible linter that can be integrated into IDEs (e.g., IntelliJ, Eclipse, and NetBeans) and build tools (e.g., Maven and Gradle).

## 2.3 Usage of Checkstyle in the wild

Linters have been the subject of investigation in recent research (Zampetti et al. 2017; Vassallo et al. 2018; Marcilio et al. 2019). However, the existing studies did not investigate at scale and look into how style checking tools are specifically used. In this section, we present a study focused on the usage of Checkstyle, which is a popular linter for Java that checks source code style.

### 2.3.1 Checkstyle usage in open-source projects

**Method.** To measure the usage of Checkstyle, we queried GitHub<sup>10</sup> to only retrieve Java projects with at least five stars, because stars have been shown meaningful to sample projects from GitHub (Beller et al. 2017). We found 171,195 Java projects. Then, we searched each of them<sup>11</sup> for finding a Checkstyle ruleset file. Note that a Checkstyle ruleset file can have any name, but we followed a conservative approach towards identifying true positive files by using a set of commonly used names<sup>12</sup>. For simplicity, hereafter, we refer to a Checkstyle ruleset file as `checkstyle.xml`.

**Results.** We found 4,334 Java projects containing a `checkstyle.xml` file, which is 2.53% of all Java projects with at least five stars on GitHub. Table 1 shows the proportion of these projects that use Maven, Gradle, or Ant as their build tools, and the Travis or Circle CI services. We note that build tools are widely used among projects using Checkstyle: 98% of the projects use at least one build tool. Moreover, 44% of the projects use a continuous integration service, which shows the software engineering maturity of the sampled projects.

<sup>5</sup><https://eslint.org/>, last access: 2020-07-13

<sup>6</sup><https://www.pylint.org/>, last access: 2020-07-13

<sup>7</sup><https://github.com/StyleCop/StyleCop/>, last access: 2020-07-13

<sup>8</sup><https://docs.rubocop.org>, last access: 2020-07-13






<sup>9</sup><https://checkstyle.sourceforge.io/>, last access: 2020-07-13

<sup>10</sup>On June 23, 2021.

<sup>11</sup>On June 23–24, 2021.

<sup>12</sup>Commonly used names for Checkstyle ruleset files: ‘checkstyle.xml’, ‘.checkstyle.xml’, ‘checkstyle.rules.xml’, ‘checkstyle.config.xml’, ‘checkstyle.configuration.xml’, ‘checkstyle\_checker.xml’, ‘checkstyle\_checks.xml’, ‘google\_checks.xml’, ‘sun\_checks.xml’. Variants by replacing ‘.’ by ‘-’ were also used.

**Table 1** Usage of build tools and CI services in the 4,334 projects using Checkstyle

Build tool	Maven	54%	
	Gradle	47%	
	Ant	9%	
CI service	Travis CI	41%	
	Circle CI	4%	

### 2.3.2 Popularity of checkstyle rules

**Method.** To check the usage of the 182 Checkstyle rules<sup>13</sup>, we analyzed the previously-found `checkstyle.xml` files from the 4,334 projects. Our goal is to investigate the most used rules and check if formatting-related rules, which are the target of this work, are widely used.

**Results.** We found out that all Checkstyle rules are used. Figure 2 shows the top-10 most used rules. The bars in dark gray represent formatting-related rules, and the bars in light gray represent non-formatting rules. In addition, the bar in gray with a dot pattern represents a rule that can be about formatting, but it depends on how it is configured since it is a regex rule. In the top-10 most used rules, there are three rules related to formatting and one that can be. Notably, the top-3 most used rules are or can be formatting-related ones. Therefore, we conclude that formatting-related rules are important for developers, which validates the relevance of our work.

## 3 STYLER

STYLER is a tool dedicated to helping developers keep their source code compliant with their adopted formatting conventions by automatically fixing formatting violations in Java source code. STYLER could be used in different software development workflows. For instance, STYLER could be used locally as a pre-hook commit when developers are about to release projects. It could also be used in continuous integration environments, where pull requests could be automatically opened with formatting fixes' suggestions. In this section, we present the workflow and the technical principles of STYLER.

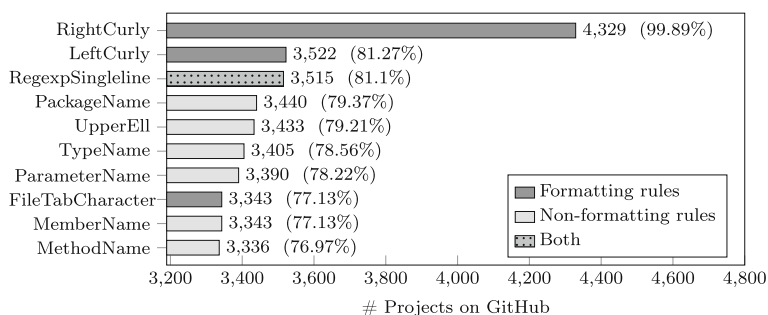
### 3.1 Targeted violation types

STYLER is about learning and repairing violations related to *formatting conventions*. For instance, consider that a developer specified that left curly braces must always be placed at the end of lines (as shown in Fig. 1b). If this rule is not satisfied (e.g., such as in Fig. 1a), a given linter triggers a formatting-related violation: for instance, Checkstyle would output the violation presented in Fig. 4a, and SonarJava<sup>14</sup> would find a violation of the rule “An open curly brace should be located at the end of a line”<sup>15</sup>. In order to fix this violation, the line break before the token “{” must be replaced by a single space.

<sup>13</sup>The set of Checkstyle rules considered in our study is from the Checkstyle version 8.43 (released on May 30, 2021).

<sup>14</sup><https://github.com/SonarSource/sonar-java>, last access: 2020-07-13

<sup>15</sup><https://rules.sonarsource.com/java/RSP-1105>, last access: 2020-07-13

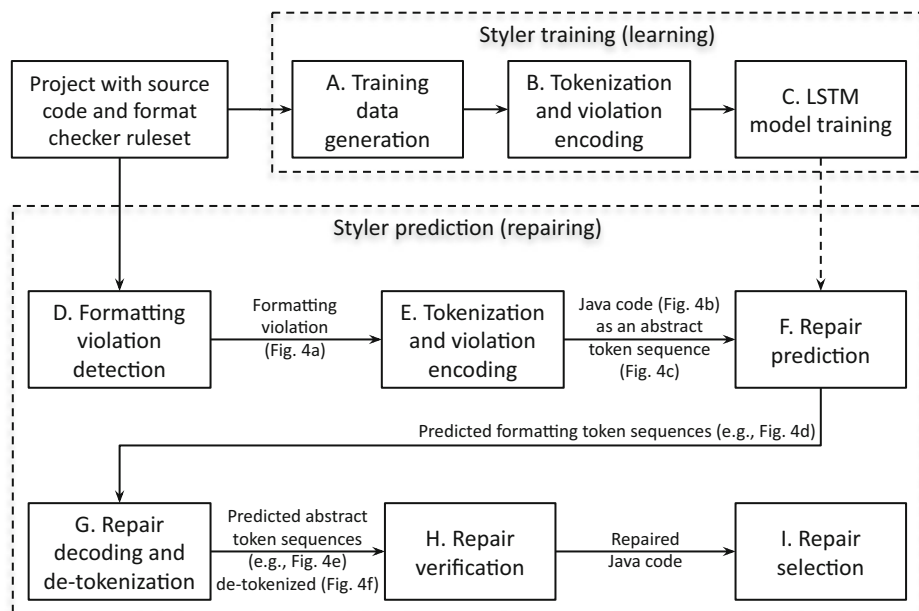


**Fig. 2** The top-10 most used Checkstyle rules

As mentioned in Section 2.1, there are different classes of conventions, e.g., formatting and naming, and consequently different automated checks in linters. In STYLER, we exclusively focus on *formatting checks related to non-printable characters*, such as indentation and whitespace before and after punctuation. Hereafter, we refer to the linter part related to these formatting checks as *format checker*.

### 3.2 STYLER workflow

Figure 3 shows the workflow of STYLER. It is composed of two main components: ‘STYLER training’ for learning how to repair formatting violations and ‘STYLER prediction’ for repairing a real formatting violation raised by a format checker. STYLER receives as input a software project, including its source code and its format checker ruleset.



**Fig. 3** STYLER workflow

The component ‘STYLER training’ is responsible for learning how to repair formatting violations on the given project according to the project-specific format checker ruleset. It creates the training data by injecting formatting violations into violation-free source code files belonging to the project (step A). Then, it encodes the training data into abstract token sequences (step B) in order to train LSTM neural networks (step C). The learned LSTM models are later used to predict repairs.

The component ‘STYLER prediction’ is responsible for predicting fixes for real formatting violations. It first detects formatting violations by running the format checker on the project (step D). Then, STYLER encodes the violating code into an abstract token sequence (step E), which is given as input to the LSTM models (step F) previously learned. The models predict fixes for the given formatting violation. These fixes are in the format of formatting token sequences, so they are translated back to Java code (step G). STYLER then runs the format checker on the new Java code containing the predicted fixes (step H). Finally, among the predicted fixes where no violation is raised by the format checker, STYLER selects one formatting fix to give as output (step I). As STYLER only impacts the formatting of source code, its repairs do not change the behavior of the program under consideration.

### 3.3 STYLER in action

Consider the formatting violation presented in Fig. 4a. This violation is about the Checkstyle *LeftCurly* rule, which was configured to enforce that left curly braces are placed at the end of lines. The Java source code that caused such a violation is presented in Fig. 4b.

```
[ERROR] DSAEncoder.java:7:1: '{' at column 1 should be on the previous line.
[LeftCurly]
```

(a) Checkstyle *LeftCurly* rule violation.

```
6 public interface DSAEncoder
7 {
8     byte[] encode(BigInteger r, BigInteger s)
```

(b) Source code snippet with the violation.

```
before-context-tokens <LeftCurly> Identifier 1_NL { 1_NL_4_ID_SP </LeftCurly>
after-context-tokens
```

(c) Original abstract token sequence.

```
1_SP 1_NL_4_ID_SP
```

(d) Formatting token sequence generated by a LSTM model.

```
before-context-tokens <LeftCurly> Identifier 1_SP { 1_NL_4_ID_SP </LeftCurly>
after-context-tokens
```

(e) Predicted abstract token sequence.

```
6 public interface DSAEncoder {
7     byte[] encode(BigInteger r, BigInteger s)
```

(f) Source code snippet with repaired formatting.

**Fig. 4** STYLER from a Checkstyle formatting violation to a fix

For that violation, STYLER encodes the incorrectly formatted lines (Fig. 4b) into the abstract token sequence shown in Fig. 4c. Then, this abstract token sequence is given as input to LSTM models, which predict alternative formatting token sequences, as the one shown in Fig. 4d, that may fix the current formatting violation. These predicted formatting token sequences are then used to modify the formatting tokens of the original abstract token sequence. It results in predicted abstract token sequences, as the one shown in Fig. 4e. The difference between Fig. 4c and 4e is the replacement of the formatting token `1_NL` by `1_SP`. This predicted repair means that the line break before the token “{” should be replaced by a single space. Then, the predicted abstract token sequence (Fig. 4e) is translated back to Java code (Fig. 4f). Finally, when running Checkstyle on the new Java code, no Checkstyle violation is raised, meaning that STYLER successfully repaired the violation.

### 3.4 Java source code encoding

STYLER encodes Java source code into an abstract token sequence that is required to predict formatting changes. An abstract token sequence is composed of pairs of abstract Java tokens and abstract formatting tokens. STYLER represents each *Java token* as an abstract token by keeping the value of the Java keywords, separators, and operators (e.g., `+`  $\rightarrow$  `+`), and by replacing the other token kinds such as literals, comments, and identifiers by their types (e.g., `x`  $\rightarrow$  `Identifier`). For each pair of subsequent Java tokens, STYLER creates an abstract *formatting token*, which depends on the presence of a new line. If there is no new line, STYLER counts the number of whitespace characters, and then represents it as `n_SP` when the characters are spaces and `n_TB` when the characters are tabulations, where `n` is the number of whitespaces characters (e.g., `_`  $\rightarrow$  `1_SP`). If there is no whitespace between two Java tokens (e.g., `x=`), STYLER adds `0_None` between the two Java tokens.

If there are new lines between two Java tokens, STYLER first counts the number of new lines and represents it as `n_NL`, where `n` is the number of new lines. Then, STYLER calculates the indentation delta ( $\Delta$ ), i.e., the indentation difference, between the line containing the first Java token and the line containing the second Java token. Positive indentation deltas are represented by  `$\Delta\_ID$`  (indent), negative ones are represented by  `$\Delta\_DD$`  (dedent), and deltas equal to zero, i.e., no indentation change between the lines, are represented by the absence of an indentation delta representation. The complete representation after the calculation of the number of new lines and the indentation delta is  `$n\_NL\_ \Delta\_ (ID|DD) \_ (SP|TB)$` . For instance, in Fig. 4b, the new line between lines 7 and 8 is represented by `1_NL_4_ID_SP`, i.e., one new line and indentation delta +4.

### 3.5 Self-supervised training data generation

STYLER does not use predefined templates for repairing formatting violations. STYLER uses machine learning for inferring a model to repair formatting violations and, consequently, it needs training data. One option would be to mine past commits from the project under consideration to collect training data. However, there might not exist enough data in the history of the project for training models.

Therefore, to have enough data for training, our key insight is to generate the training data in a self-supervised manner. The idea is to modify violation-free Java files belonging to the project under analysis to trigger formatting rule violations. A similar idea has been explored by Yasunaga and Liang (2020). Then, one obtains a pair of files ( $\alpha_{orig}$ ,  $\alpha_{err}$ ):  $\alpha_{orig}$  is the file without the formatting violation, and  $\alpha_{err}$  is the file with the formatting violation.  $\alpha_{orig}$  is a repaired version of  $\alpha_{err}$ , and we can use supervised machine learning

to predict  $\alpha_{orig}$  given  $\alpha_{err}$ . We explore this idea in two different ways to generate training data, hereafter referred to as formatting violation injection *protocols*. The protocol names are *Styler<sub>random</sub>* and *Styler<sub>3grams</sub>*.

The *Styler<sub>random</sub>* protocol for injecting formatting violations in a project consists of automated insertion or deletion of a single formatting character (space, tabulation, or new line) in Java source files. These modifications require a careful procedure so that 1) the project still compiles and 2) its behavior is not changed. For this, we specify the locations in the source code files that are suitable to perform the modifications. For insertions, the suitable locations are before or after any token. For deletions, the suitable locations are 1) before or after any punctuation (“.”, “;”, “(”, “)”, “[”, “]”, “{”, “}”, and “;”), 2) before or after any operator (e.g., “+”, “-”, “\*”, “=”, “+=”), and 3) in any token sequence longer than one indentation character.

The *Styler<sub>3grams</sub>* protocol is meant to produce likely violations. Instead of directly changing the Java source code as *Styler<sub>random</sub>*, *Styler<sub>3grams</sub>* performs modifications at the abstract token level. The idea is to replace formatting tokens with the ones used by developers in the same context, i.e., between the same surrounding Java tokens. For that, we use 3-grams, where a 3-gram = {Java token, formatting token, Java token}. So given a violation-free Java file, the task of *Styler<sub>3grams</sub>* is the following. First, the Java file is tokenized (see Section 3.4), and a random formatting token is picked and used to form a 3-gram, which is 3-gram<sub>orig</sub>. Then, given a corpus of 3-grams previously created from software projects, *Styler<sub>3grams</sub>* finds a 3-gram<sub>i</sub> in the corpus that matches the Java tokens of 3-gram<sub>orig</sub>. Several matches can be found, but the selection of a 3-gram<sub>i</sub> is random according to its frequency in the corpus. Then, the formatting token of 3-gram<sub>orig</sub> is replaced by the formatting token of 3-gram<sub>i</sub>. Finally, *Styler<sub>3grams</sub>* performs a de-tokenization so that a violating Java version of the original violation-free Java file is created.

Algorithm 1 presents the algorithm that STYLER uses to generate one training dataset per formatting violation injection protocol (*Styler<sub>random</sub>* and *Styler<sub>3grams</sub>*). The input of the algorithm is the format checker ruleset of the project, a corpus of violation-free Java files taken from the project, the number of violating files to be generated, the injection protocol to be used, and the maximum duration of the process. Then, in each batch iteration (line 7), a file is randomly selected from the corpus of violation-free Java files (line 12), and the specified injection protocol is applied to it (line 13). Once a batch is completed, the format checker is executed on the resulting modified files (line 16) so that the algorithm selects the ones that contain a single violation (line 17). The algorithm ends when the desired number of files with violations is reached or when the process reaches the specified maximum duration.

### 3.6 Violation encoding

In order to repair formatting violations, the Java source code encoded as an abstract token sequence must capture both the violation in the code and the context surrounding the violation. So, for a given violation, STYLER considers a token window of  $k$  source code lines before and after the violation location provided by the format checker for creating an abstract token sequence (see Section 3.4). Once the violating line and the ones surrounding it are tokenized, STYLER places two tags around the tokens related to the origin of the violation so that the violation location and its type can be further identified. The tags consist of the name of the format checker rule that was violated. For instance, the violation presented in Fig. 4a is about the Checkstyle *LeftCurly* rule, so the tags around the violation are <LeftCurly> and </LeftCurly> as shown in Fig. 4c.

**Algorithm 1** Injection of formatting violations in Java files.

---

**Input:** *ruleset* – format checker configuration of the project under consideration  
**Input:** *files* – corpus of violation-free Java files taken from the project  
**Input:** *numberOfViolations* – number of files with one violation to be generated  
**Input:** *protocol* in [*Styler<sub>random</sub>*, *Styler<sub>3grams</sub>*]  
**Input:** *maxDuration* – maximum duration of the process  
**Output:** *dataset* with Java files containing formatting violations raised by a format checker

```

1: var dataset  $\leftarrow \{\}$ 
2: var maxTime  $\leftarrow \text{time.now} + \text{maxDuration}$ 
3: var timeout  $\leftarrow \text{false}$ 
4: while dataset.size < numberOfViolations and not timeout do
5:   var modifiedFiles  $\leftarrow \{\}$ 
6:   batchSize  $\leftarrow \text{numberOfViolations} - \text{dataset.size}$ 
7:   for i  $\leftarrow 0; i < \text{batchSize}; i++$  do
8:     if time.now  $\geq \text{maxTime}$  then
9:       timeout  $\leftarrow \text{true}$ 
10:      go to line 16
11:    end if
12:    file  $\leftarrow \text{selectRandomly}(\text{files})$ 
13:    file'  $\leftarrow \text{changeFormatting}(\text{file}, \text{protocol})$ 
14:    modifiedFiles.append(file')
15:  end for
16:  formatCheckerResults  $\leftarrow \text{runFormatChecker}(\text{modifiedFiles}, \text{ruleset})$ 
17:  violatingFiles  $\leftarrow \text{selectFilesWithOneViolation}(\text{formatCheckerResults})$ 
18:  dataset.append(violatingFiles)
19: end while
20: return dataset

```

---

The strategy to place the tags in the abstract token sequence is primarily based on the fact that the tags should surround the tokens related to the origin of the violation. At the same time, the number of tokens between the two tags should be minimal so to keep precise information about the violation location. Thus, STYLER places the tags according to the location information given by the format checker. When the format checker provides the line and the column, STYLER places `<ViolationType>` one token before the violation and `</ViolationType>` one token after. When the format checker provides the line but not the column (e.g., when the violation is about the Checkstyle *LineLength* rule), STYLER places `<ViolationType>` one token before the line and `</ViolationType>` one token after the end of the line.

### 3.7 Machine learning model

**Learning (Fig. 3–step C).** STYLER aims to translate a token sequence with a formatting violation (input sequence) to a new token sequence with no formatting violation (output sequence). STYLER uses a sequence-to-sequence translation based on recurrent neural network LSTMs (Long Short-Term Memory), similar to what is used for natural language translation. Thanks to the token abstraction employed by STYLER to encode Java source code (see Section 3.4 and Section 3.6), the input and output vocabularies are small



single violation. After the translation of these predictions back to Java source code (Fig. 3–step G), STYLER performs a verification (Fig. 3–step H), where the format checker is executed on the resulting Java source code files. Finally, given the files that do not result in formatting violations, STYLER selects the one that has the smallest source code diff to give as output (Fig. 3–step I).

### 3.9 Implementation

The approach employed by STYLER is independent of the considered format checker. The current implementation uses Checkstyle, which is a popular format checker for Java. Other format checkers can be integrated in STYLER. However, they must output the violation type and the violation location. This is necessary for the violation encoding (see Section 3.6).

STYLER is implemented in Python. We use javalang<sup>16</sup> for parsing and OpenNMT-py<sup>17</sup> for the machine learning part. STYLER is publicly available at <https://github.com/KTH/styler/>. The current calibration of STYLER is presented in Section 4.4.1.

## 4 Evaluation design

We conducted an empirical study to evaluate STYLER from different perspectives (see Section 4.1), including a comparison against three state-of-the-art code formatting systems (see Section 4.2). We first built a dataset of Checkstyle violations mined from GitHub repositories (see Section 4.3), and then we gave these violations as input to all the four tools (see Section 4.4) to measure their reparability. In this section, we present the design of our study.

### 4.1 Research questions

Our goal is to answer the following six research questions.

**RQ #1** [Overall reparability]: To what extent does STYLER repair real-world Checkstyle formatting violations, compared to other systems?

Overall reparability is an important metric to measure the value of tools. We investigate the reparability of STYLER on real Checkstyle violations, which allows us to understand to what extent STYLER repairs formatting violations that have occurred in practice. Moreover, we compare the reparability of STYLER to the reparability of three code formatters by using the same dataset of violations to investigate if, and to what extent, STYLER outperforms the related systems.

**RQ #2** [Violation-type-based reparability]: To what extent does STYLER repair different violation types, compared to other systems?

Checkstyle has different formatting rules, so it raises different violation types. In this research question, we investigate if, and to what extent, STYLER repairs different violation types compared to the other systems. This analysis is also important to investigate if the systems are complementary to each other.

<sup>16</sup><https://github.com/c2nes/javalang/>, last access: 2020-07-13

<sup>17</sup><https://github.com/OpenNMT/OpenNMT-py/>, last access: 2020-07-13

**RQ #3** [Unsuccessful repair cases]: What are the cases in which STYLER fails to generate a correct repair?

Understanding the cases in which STYLER fails to generate a correct repair is important so that i) STYLER can be further improved, ii) hard-to-repair violations are identified and, consequently, researchers might study them and develop tools specialized to repair them, and iii) the limitations of STYLER can be taken into account by developers when deciding whether or not to use STYLER in their projects. To discover the most frequent cases in which STYLER does not succeed to generate a correct repair, we manually analyzed violations of the rules for which STYLER does not perform well.

**RQ #4** [Quality]: What is the size of the repairs generated by STYLER, compared to other systems?

What are the cases in which STYLER fails to generate a correct repair? There may be several alternative repairs that fix a given Checkstyle violation, including ones that change source code lines other than the ill-formatted line. In this research question, we compare the size of the repairs generated by STYLER against the repairs generated by the other systems.

**RQ #5** [Performance]: How fast is STYLER for learning and predicting formatting repairs?

To investigate if STYLER could be applicable in practice, we measure its performance for fixing the mined Checkstyle violations. This is valuable information for those who could be interested in using STYLER as a pre-commit hook in IDEs or continuous integration services.

**RQ #6** [Technical analysis]: How do the two training data generation techniques of STYLER contribute to its repairability?

Finally, we perform a comparison between the two formatting violation injection protocols used to generate training data (see Section 3.5). This comparison is done through the LSTM models trained with the two different training sets. We investigate if, and to what extent, one of the models contributes more to the repairability of STYLER. This is an important investigation from the point of view of users who might want to use STYLER with only one model for performance reasons.

## 4.2 Systems under comparison

We selected three systems to compare STYLER with: CHECKSTYLE-IDEA (CheckStyle-IDEA 2021), NATURALIZE (Allamanis et al. 2014), and CODEBUFF (Parr and Vinju 2016). CHECKSTYLE-IDEA, also referred to as CS-IDEA in this paper, is an IDE-based code formatter plugin for the IntelliJ IDE. It provides IDE-integrated feedback against a given Checkstyle ruleset and fixes Checkstyle violations through the IntelliJ formatter by taking into consideration a Checkstyle ruleset. NATURALIZE is a tool dedicated to assisting developers in fixing coding conventions related to naming and formatting in Java programs. It learns coding conventions from a codebase and suggests fixes to developers, such as formatting modifications, based on the n-gram model. CODEBUFF is a code formatter applicable to any programming language with an ANTLR grammar. Instead of formatting the code according to ad-hoc rules for a language, CODEBUFF aims to infer the formatting rules given a grammar for the language and a set of files following the same formatting rules. For

each token, a KNN model decides to indent it or to align it with another token based on the abstract syntax tree of the source file.

All three systems are code formatters. CHECKSTYLE-IDEA takes a Checkstyle ruleset into consideration, and NATURALIZE and CODEBUFF are the state-of-the-art machine learning formatters that aim to assist developers to fix formatting-related issues without any prior or ad-hoc formatting rules.

### 4.3 Data collection

To execute STYLER and the systems under comparison and, consequently, answer our research questions, we created a dataset of Checkstyle formatting violations by mining open-source projects. The first step was to build a list of projects, which was done based on the data previously collected for the study presented in Section 2.3. We selected all the projects that have exactly one Checkstyle ruleset file and use Maven. This resulted in 2,143 projects.

For each project, we tried to reproduce Checkstyle violations with the following automated lightweight approach. First, the remote repository of the project is cloned from GitHub<sup>18</sup>. Then, a sanity check is performed on the `checkstyle.xml` file contained in the project. If the file contains variables, the project is discarded. Otherwise, a search in the history of the project is done for the last commit ( $c_i$ ) that contains modifications in the `checkstyle.xml` file, which is the commit to be used as the starting point for the reproduction of real violations. Then,  $c_i$  is checked out, and all the files of the project are submitted to a process that aims to check if our automated approach can successfully execute Checkstyle on the project and with which version of Checkstyle. The latter is necessary because new versions of Checkstyle might introduce breaking backward compatibility<sup>19</sup> and, then, fail to parse a `checkstyle.xml` file that was used with previous versions of Checkstyle. Such a process consists of executing multiple Checkstyle versions on the project, from a newer version to an older one, until finding one version that does not fail or until the available options end<sup>20</sup>. If a successful Checkstyle execution is found, the last tested Checkstyle version,  $x$ , is chosen to be used on the project. All commits since  $c_i$  are then gathered, inclusive, so that all commits to be analyzed are based on the same Checkstyle ruleset.

Then, each selected commit is checked out, and a sanity check is performed on the `pom.xml` file of the commit. If it points to a suppression file, the commit is discarded because we want violations that happened in practice and our lightweight approach does not solve paths. Otherwise, the Checkstyle version  $x$  is executed on the files of the project. If the commit under analysis is the first one to be analyzed, Checkstyle is executed on all the files of the project. Otherwise, Checkstyle is executed only on the files changed in the commit under analysis to avoid duplicate violations in the dataset. Moreover, Java files in folders named *test* or *resources* are ignored since we want violations that happened in the main source code. Then, after executing Checkstyle, if at least one Checkstyle violation is raised, the violating Java files and information about the violations, e.g., Checkstyle violation types and locations, are saved.

<sup>18</sup>All repositories were cloned on June 24–25, 2021.

<sup>19</sup>Checkstyle release notes: <https://checkstyle.sourceforge.io/releasenotes.html>

<sup>20</sup>Our current implementation supports 48 Checkstyle versions, from 8.0 to 8.43.

Such a process was executed for all the 2,143 projects in our list. At the end of the process, we removed duplicate Java files according to the file content among all commits if any. Then, we selected the files containing a single Checkstyle violation that is related to formatting. We performed this selection to accurately evaluate repairs produced by STYLER and the other tools. Finally, we kept projects where all criteria yield at least 20 Checkstyle formatting violations. By applying this systematic reproduction and selection process, we obtained a dataset containing 27,058 Checkstyle violations spread over 105 projects. We used one project, and the violations found in it (267), to calibrate STYLER (see Section 4.4.1), and the other 104 projects with 26,791 violations for the actual evaluation.

The dataset is diverse in terms of projects and violation types. Table 2 shows the subject projects. The projects are very diverse in several aspects, such as in number of Java files ( $min = 2$ ,  $med = 532$ ,  $max = 18,206$ ), commits ( $min = 13$ ,  $med = 1,251$ ,  $max = 71,544$ ), contributors ( $min = 1$ ,  $med = 14$ ,  $max = 249$ ), and stars ( $min = 5$ ,  $med = 46$ ,  $max = 24,888$ ). Additionally, Table 3 shows the stats per Checkstyle formatting rule. We note that the most frequent violations in our dataset are violations of the rules *RegexpSingleline*, *EmptyLineSeparator*, and *LineLength*.

## 4.4 Setup and execution of the systems

We gave the dataset of violations as input to STYLER and the three systems under comparison to evaluate their repairability. In this section, we present the setup of the systems, which includes the calibration of STYLER, the adaptations performed in NATURALIZE and CODEBUFF, and how the four systems were executed.

### 4.4.1 STYLER calibration

To calibrate STYLER, i.e., the *Styler<sub>random</sub>*- and *Styler<sub>3grams</sub>*-based models, we performed an exploratory study by training LSTM models with different configurations. The configurations combine values for key parameters, which are the model attention type (general or mlp), the number of layers (one or two) and units (256 or 512) for the model encoder/decoder, and the model word embedding size (256 or 512). For each configuration, the training was performed for a maximum of 20k iterations, with a batch size of 32, and a model was saved in the iterations 5k, 10k, 15k, and 20k. This means that, in the end, we obtained 64 models (2 model attention types  $\times$  2 numbers of layers  $\times$  2 numbers of units  $\times$  2 embedding sizes  $\times$  4 training iterations) per training data generation protocol (i.e., *Styler<sub>random</sub>* and *Styler<sub>3grams</sub>*).

Those models were created for one open-source project<sup>21</sup> contained in our dataset (see Section 4.3), which was randomly selected from the top-5 projects with the most diversity of violated formatting rules. The project was given as input to STYLER, which produced training data by injecting Checkstyle violations in violation-free files found in the project (see Section 3.5). For each protocol, 10k violations were injected. This data was used to train the LSTM models, where 9k violations were used for training and 1k for validation. Once the 64 models per protocol were created, we executed STYLER with each of them on the real violations found in the project so that we could test the models and choose the configuration of the best ones. Then, for each protocol, we picked the configuration of the model that repaired violations in a more balanced way in terms of Checkstyle rules. The

<sup>21</sup><https://github.com/inovexcorp/mobi>

**Table 2** Subject projects<sup>a</sup>. 104 projects were used to evaluate STYLER and one project (flagged with “★”) was used to calibrate STYLER

Projects (104 + 1)	# Java files	# Commits	# Contributors	# Stars	# Violations
landl/cosmo	557	594	8	54	26
actiontech/txle	208	1,416	44	35	49
Activiti/Activiti	1,532	10,758	192	7,837	1,032
Angel-ML/angel	861	2,714	47	6,282	1,143
apache/crunch	522	1,093	34	100	26
apache/ignite-3	344	201	23	23	80
apache/iotdb	1,215	5,001	121	1,388	491
apache/metron	568	1,453	60	828	115
apache/servicecomb-java-chassis	907	2,997	98	1,703	398
apache/shardingsphere	1,441	29,153	249	14,097	43
apache/usergrid	639	10,954	78	992	145
aspose-words- cloud/aspose-words- cloud-java	313	325	8	14	594
atlanmod/NeoEMF	293	2,998	10	38	87
bakdata/conquery	628	7,831	17	25	884
benetech/ServiceNet	660	1,618	12	7	69
black-ant/case	1,899	353	2	31	46
blockchain- lab/ScaleOutDistributedLedger	2	574	6	10	29
ByteHamster/PSE	133	835	5	8	29
CESNET/perun	1,500	7,359	42	46	525
Chaklader/Multi-threading	532	55	1	16	35
Chaklader/Object-Oriented-Design	2985	35	1	240	67
chenpengliang0909/WxJava	622	2,209	73	13	30
ciandt-dev/tech-gallery	122	1,312	20	38	20
cloudera/director-sdk	516	39	7	20	317
codefollower/H2-Research	656	12,913	114	387	573
CONNECT-Solution/CONNECT	1,860	11,828	33	65	1,463
couchbase/couchbase-java-client	164	1,051	22	257	28
couchbase/couchbase-jvm-core	242	968	17	31	98
ctripcorp/apollo	78	2,526	78	24,888	144
DaGeRe/peass	103	907	1	7	114
decorators-squad/eo-yaml	58	834	12	203	39
decorators-squad/versioneye-api	4	127	3	7	57
delight-im/NationSoccer	847	18	1	13	20
DigitalMediaServer/DigitalMediaServer	502	7,315	38	24	42
DSC-SPIDAL/harp	580	900	18	20	50
dzhw/metadatamanagement	528	8,979	18	21	163
eclipse-ee4j/glassfish	12,820	1,337	48	233	140
eclipse/milo	1,434	1,119	16	618	273

**Table 2** (continued)

Projects (104 + 1)	# Java files	# Commits	# Contributors	# Stars	# Violations
EMResearch/EMB	1,461	268	3	5	112
fangjinuo/easyjson	91	624	3	58	68
farao-community/farao-core	230	364	17	7	51
findbugsproject/findbugs	2,310	15,375	35	661	35
GluuFederation/casa	140	698	9	9	61
GluuFederation/oxCore	353	1,457	17	11	201
gomint/gomint	2953	1,668	26	221	42
googleapis/google-cloud-java	65	5,128	200	1,559	843
GourdErwa/MyNote	1,151	53	1	72	57
graphfoundation/ongdb	4,587	220	2	211	4,569
griddynamics/jagger	870	2,343	25	65	184
Gurux/gurux.dlms.java	175	377	2	52	31
h2database/h2database	656	13,570	127	2,849	215
HealerJean/HealerJean.github.io	2,752	1,385	2	23	112
HuygensING/timbuctoo	621	8,311	14	38	66
ibinti/bugvm	9,665	3,647	23	348	171
★ inovexcorp/mobi	599	12,454	19	35	267
Internet2/grouper	4,661	10,368	22	59	101
intuit/Tank	1,160	320	7	61	217
IQSS/dataverse	807	19,817	119	600	132
ita-social-projects/GreenCity	182	2,164	38	41	29
java110/MicroCommunity	711	4,172	12	516	584
jflex-de/jflex	126	1,874	11	400	53
junkdog/artemis-odb	266	1,754	26	638	111
kitodo/kitodo-production	602	12,109	30	47	43
ldp4j/ldp4j	752	459	2	43	40
liuawen/Learning-Java	18,206	312	2	184	20
LoboEvolution/LoboEvolution	1,712	416	1	25	580
ManfredTremmel/gwt-bean-validators	934	219	3	26	23
matsim-org/matsim-episim-lib	59	2,528	18	15	33
NationalSecurityAgency/datawave	1,939	2,343	42	360	140
NationalSecurityAgency/emissary	317	186	16	171	20
neo4j/neo4j	3,641	71,544	210	9,060	3,375
neuhajje/bouncy-gpg	44	440	8	168	44
O2-Czech-Republic/proxima-platform	314	1,611	7	12	91
omnisci/omnisci-jdbc	92	76	12	6	61
open-eid/digidoc4j	287	2,000	25	55	95
opencb/opencga	837	12,047	39	120	410
OpenEMS/openems	1,615	4,962	32	194	46
openmessaging/openmessaging-java	55	332	11	696	61
Plugily-Projects/MurderMystery	66	694	15	105	105

**Table 2** (continued)












































































Projects (104 + 1)	# Java files	# Commits	# Contributors	# Stars	# Violations
ppati000/visualDFA	18	270	6	8	26
primefaces/primefaces	1,250	12,177	160	1,367	33
Qihoo360/Quicksql	1,329	191	11	1,733	109
Roboy/roboy_dialog	419	1,251	17	9	26
Rugal/algorithm	97	421	1	6	51
RWTH-i5-IDSG/steve	185	1,375	11	215	174
seedstack/business	252	309	7	20	196
seedstack/seed	439	698	10	24	351
self-xdsd/self-core	222	1,488	11	19	52
self-xdsd/self-storage	21	384	5	11	35
SentinelDataHub/dhus-core	412	13	3	6	35
SergeyZhernovoy/Java-education	265	441	1	8	40
spark-root/laurelin	44	440	4	7	30
StevenLooman/sonar-magik	54	178	2	5	27
Stratio/bdt	54	1,023	34	65	70
StuPro-TOSCA/TOSCAAna	122	606	6	10	33
TIBCOSoftware/genxadm	1,025	930	5	6	501
tmobile/kardio	206	41	9	149	130
toast-tk/toast-tk-engine	207	418	5	12	41
twilio/twilio-java	654	2,028	93	377	1,126
V1toss/JavaPA	146	242	1	16	28
vassalengine/vassal	986	7,861	14	99	28
vostok/hercules	322	1,988	9	16	118
wayshall/onetwo	1,221	4,690	2	17	158
wso2-attic/commons	7,399	904	1	10	587
zanata/zanata-platform	1,743	19,064	29	288	70
min	2	13	1	5	20
med	532	1,251	14	46	70
max	18,206	71,544	249	24,888	4,569

<sup>a</sup>The numbers of commits, contributors, and stars were retrieved on June 23–24, 2021. The number of violations is the number of formatting-related violations found in the project and that were selected according to our criteria

best *Styler<sub>random</sub>*-based model was with the mlp model attention, one layer, 256 units, embedding size of 512, and 5k training iterations, and the best *Styler<sub>3grams</sub>*-based model was with the same values for the numbers of layers, embedding size, and training iterations, but with the general model attention and 512 units. These are the configurations we used for training the models for our experiments presented in Section 5.

For prediction, the beam search creates  $x = 5$  potential repairs per model. As for the violation encoding (see Section 3.6), we set  $k = 6$ . Recall that this parameter is about the token window before and after the violation (i.e., the context surrounding the violation).

**Table 3** Checkstyle formatting violation dataset stats per rule<sup>a</sup>

Checkstyle rule (25)	Calibration		Evaluation	
	Violations (267)	Violations (26,791)	Projects (104)	
CommentsIndentation	12 ( 4%) 	30 (<1%) 	10 (10%) 	
EmptyForIteratorPad	0 ( 0%) 	11 (<1%) 	2 ( 2%) 	
EmptyLineSeparator	11 ( 4%) 	4,837 (18%) 	34 (33%) 	
FileTabCharacter	4 ( 1%) 	1,807 ( 7%) 	30 (29%) 	
GenericWhitespace	3 ( 1%) 	4 (<1%) 	4 ( 4%) 	
Indentation	7 ( 3%) 	1,531 ( 6%) 	23 (22%) 	
LeftCurly	0 ( 0%) 	400 ( 1%) 	31 (30%) 	
LineLength	142 (53%) 	4,270 (16%) 	59 (57%) 	
MethodParamPad	0 ( 0%) 	86 (<1%) 	13 (13%) 	
NoLineWrap	0 ( 0%) 	30 (<1%) 	2 ( 2%) 	
NoWhitespaceAfter	0 ( 0%) 	59 (<1%) 	9 ( 9%) 	
NoWhitespaceBefore	0 ( 0%) 	179 ( 1%) 	20 (19%) 	
OneStatementPerLine	0 ( 0%) 	4 (<1%) 	2 ( 2%) 	
OperatorWrap	10 ( 4%) 	274 ( 1%) 	26 (25%) 	
ParenPad	0 ( 0%) 	110 (<1%) 	12 (12%) 	
Regexp	0 ( 0%) 	688 ( 3%) 	8 ( 8%) 	
RegexpMultiline	0 ( 0%) 	15 (<1%) 	1 ( 1%) 	
RegexpSingleline	0 ( 0%) 	8,678 (32%) 	20 (19%) 	
RegexpSinglelineJava	0 ( 0%) 	588 ( 2%) 	4 ( 4%) 	
RightCurly	57 (21%) 	540 ( 2%) 	23 (22%) 	
SeparatorWrap	3 ( 1%) 	7 (<1%) 	3 ( 3%) 	
SingleSpaceSeparator	0 ( 0%) 	22 (<1%) 	2 ( 2%) 	
TrailingComment	0 ( 0%) 	552 ( 2%) 	5 ( 5%) 	
WhitespaceAfter	0 ( 0%) 	1,053 ( 4%) 	29 (28%) 	
WhitespaceAround	18 ( 7%) 	1,016 ( 4%) 	50 (48%) 	

<sup>a</sup>Considering Checkstyle, STYLER also targets the following rules that are not contained in our dataset: *AnnotationLocation*, *AnnotationOnSameLine*, *EmptyForInitializerPad*, and *TypecastParenPad*

This parameter is made big enough to contain important information and, at the same time, small enough to still allow learning and prediction, and was set based on meta-optimization.

#### 4.4.2 NATURALIZE and CODEBUFF adaptation

To use NATURALIZE, we had to slightly modify it. NATURALIZE recommends multiple fixes, so we take the first one for a given violation as being the repair. In addition, we changed NATURALIZE to only work for indentation, excluding fixes regarding naming conventions (which are out of the scope of this paper). To run CODEBUFF, we give it the required configuration, including the number of spaces for indentation. This value depends on the project given as input to CODEBUFF. Thus, before running CODEBUFF on a project, we count the most frequent indentation size found in the violation-free files of the project and provide it to CODEBUFF.

#### 4.4.3 Execution of the systems

The four systems were executed to repair the 26,791 violations found in the 104 projects contained in the real violation dataset. The machine-learning-based systems (STYLER, NATURALIZE, and CODEBUFF) require a corpus of violation-free files to be trained. Therefore, for each subject project, we selected, as training seeds, all violation-free Java files from

the first commit, or any subsequent one, that uses the same Checkstyle ruleset used to collect the real violations. We took special care of consistency in our experiment: all the three machine-learning-based systems were trained to repair a given project using the same corpus of violation-free files from the project.

STYLER requires other input for training. Recall that its training process includes a step for creating the actual training data (see Fig. 3—step A), which is based on the corpus of violation-free files. For each protocol, we set Algorithm 1 to create 10,000 files per project, with a maximum duration of three hours. The resulting files with violations were split for learning and validation in a balanced way according to the violation types, considering 90% for learning and 10% for validation.

Finally, to run CHECKSTYLE-IDEA on each subject project, we first loaded the violating Java files and the `checkstyle.xml` file contained in the project in IntelliJ. Then, we imported the Checkstyle ruleset (*Settings > Editor > Code Style > Import Scheme > Checkstyle Configuration*) and simply called the function “Reformat Code” from the IDE.

## 5 Evaluation results





















We present and discuss the results for our six research questions in this section.

### 5.1 Overall repairability (RQ #1)

To investigate the overall repairability of STYLER and the other three systems on the 26,791 Checkstyle violations, we categorized the repair attempts per status, as shown in Table 4. There are two groups of status: *repaired* and *not repaired*. The repaired violations are either *fully repaired*, i.e., no violation is raised after the repair attempt, or *partially repaired*, i.e., the violation no longer exists in the source code but *new violations* were introduced. For the sake of clarity, it is worth mentioning that only the full repairs are used for the other five research questions. The group of violations that were not repaired includes the cases where a resulting source code file still contains the *same violation* only or the *same + new* violations, or is *broken*, which means that the file cannot be parsed by javalang after the repair attempt.

STYLER fully repaired 41% of the violations while CS-IDEA repaired 50%, which is the greatest overall repairability among the four considered tools. NATURALIZE and CODEBUFF repaired fewer violations (15% and 20%, respectively). To check if there is a significant difference between STYLER and the other tools regarding the full repairs, we used McNemar test. Table 5 shows the contingency tables given as input to the test. We found  $p\text{-value} < 0.00001$  for all the three tests. Considering  $\alpha = 0.05$ , this means that

**Table 4** Repairability results on the 26,791 real violations per tool

Tool	Repaired violations		Not repaired violations		
	fully repaired	new violations	same violation	same + new	broken
STYLER	41% 	12% 	45% 	<1% 	2% 
CS-IDEA	50% 	10% 	12% 	28% 	0% 
NATURALIZE	15% 	15% 	28% 	8% 	34% 
CODEBUFF	20% 	17% 	10% 	12% 	41% 

**Table 5** Contingency tables of the repairability of STYLER versus the other tools





















		STYLER		McNemar test
		repaired	not repaired	p-value
CS-IDEA	repaired	9,039	4,337	<0.00001
	not repaired	1,969	11,446	
NATURALIZE	repaired	3,290	610	<0.00001
	not repaired	7,718	15,173	
CODEBUFF	repaired	3,530	1,796	<0.00001
	not repaired	7,478	13,987	

STYLER and any other tool have a statistically significant different proportion of errors on our dataset of violations. Note that the p-values were not adjusted since they are too small and the adjustment would have no impact.

Considering the numbers presented in Table 4 other than the proportions of fully-repaired violations, we noticed that CS-IDEA and STYLER are the most reliable tools in the sense of delivering to an end-user either a repaired source code or, in the worst-case scenario, the code with the same violation. It is not the same case of NATURALIZE and CODEBUFF, which had higher rates of delivering broken source code. They were, however, designed for a different goal and do not take into account the Checkstyle ruleset of the project like STYLER and CS-IDEA do. Yet, they are relevant for our experiment since they are the state of the art of machine-learning-based code formatters. Our results show the need for specialized, focused tools to repair Checkstyle violations.

In addition, we observed that some violation types, i.e., violations of different Checkstyle rules, occur in a much higher frequency than others in our dataset (see Table 3). This might cause bias in the results presented in Table 4. Because of that, we performed a normalization of the data by sub-sampling the most frequent violation types. In this way, we obtained a sub-sample of violations that contains the same number of instances for all violation types. We ignored the less frequent ones to avoid using too few instances. For that, we calculated the median of the distribution of the violation types, which is 274, and used it as the minimum number of instances for including Checkstyle rules in the analysis. Then, we randomly selected 274 violations of the included rules. In the end, the analysis comprises half of the rules (13) and 3,562 violations. The normalized results are presented in Table 6. All the tools are impacted positively in terms of fully-repaired violations. However, we note that the normalized results present a different ranking of the tools' performance, where STYLER outperforms CS-IDEA. CS-IDEA is the tool most negatively impacted by the normalization because it increases only 4% fully-repairs, while the other three tools considerably increase their repairability by 7%–20%. This suggests that CS-IDEA performs better than the other tools on violation types that are frequent in our dataset, which is investigated in more detail in the next section, for answering RQ #2. Finally, we also performed McNemar test in the normalized results, as shown in Table 7. Considering  $\alpha = 0.05$ , the results show that STYLER and any other tool have a statistically significant different proportion of errors on the sub-sample too.

**Table 6** Normalized reparability results based on 3,562 violations equally distributed across 13 violation types

Tool	Repaired violations		Not repaired violations		
	fully repaired	new violations	same violation	same + new	broken
STYLER	61% 	12% 	24% 	1% 	1% 
CS-IDEA	54% 	22% 	14% 	9% 	0% 
NATURALIZE	22% 	22% 	16% 	7% 	32% 
CODEBUFF	34% 	29% 	5% 	10% 	23% 

**RQ #1: To what extent does STYLER repair real-world Checkstyle formatting violations, compared to other systems?**

STYLER repaired 41% (11,008/26,791) of the Checkstyle formatting violations found in the wild. STYLER outperformed the machine learning systems NATURALIZE and CODEBUFF. CS-IDEA outperformed STYLER on our entire dataset of violations, with a reparability of 50%. However, this is not the case when we consider a sub-sample of the dataset by normalizing the number of instances according to violation types, which suggests that CS-IDEA performed better than the other tools on violations of frequent types in our dataset. In addition, note that CS-IDEA depends on the code formatter of the IntelliJ IDE, whereas STYLER's approach is fully automated and hence more appropriate for handling new and configurable rules.

**5.2 Violation-type-based reparability (RQ #2)**

To answer RQ #2, we investigated the extent to which STYLER and the other three systems repair different Checkstyle violation types, i.e., violations of different Checkstyle rules. Figure 6 shows the Checkstyle violations fully repaired by the systems per violation type in a heatmap. The color scale is from black to white, where black represents 0% of fully-repaired violations and white represents 100% (i.e., the lighter, the better).

STYLER and NATURALIZE repaired violations of 24/25 Checkstyle rules, which is the highest coverage of rules considering all the four tools. CS-IDEA and CODEBUFF fixed

**Table 7** Contingency tables of the reparability of STYLER versus the other tools considering the normalized results

		STYLER		McNemar test
		repaired	not repaired	p-value
CS-IDEA	repaired	1,457	479	<0.00001
	not repaired	732	894	
NATURALIZE	repaired	736	55	<0.00001
	not repaired	1,453	1,318	
CODEBUFF	repaired	937	284	<0.00001
	not repaired	1,252	1,089	

CommentsIndentation (30) –	33.3%	16.7%	0.0%	0.0%	36.7%
EmptyForIteratorPad (11) –	100.0%	0.0%	63.6%	45.5%	100.0%
EmptyLineSeparator (4837) –	83.3%	90.8%	32.7%	0.6%	99.8%
FileTabCharacter (1807) –	8.9%	95.2%	9.4%	87.7%	98.6%
GenericWhitespace (4) –	100.0%	100.0%	25.0%	75.0%	100.0%
Indentation (1531) –	88.8%	91.8%	5.0%	76.2%	96.1%
LeftCurly (400) –	98.8%	95.2%	61.5%	89.2%	100.0%
LineLength (4270) –	19.6%	55.0%	0.8%	4.4%	56.7%
MethodParamPad (86) –	93.0%	76.7%	47.7%	17.4%	93.0%
NoLineWrap (30) –	76.7%	0.0%	93.3%	0.0%	100.0%
NoWhitespaceAfter (59) –	91.5%	39.0%	74.6%	32.2%	100.0%
NoWhitespaceBefore (179) –	92.7%	63.1%	38.5%	41.9%	92.7%
OneStatementPerLine (4) –	0.0%	25.0%	25.0%	25.0%	25.0%
OperatorWrap (274) –	88.7%	0.0%	16.8%	13.1%	89.8%
ParenPad (110) –	100.0%	62.7%	50.9%	40.0%	100.0%
Regexp (688) –	48.1%	46.4%	21.2%	42.9%	53.1%
RegexpMultiline (15) –	100.0%	0.0%	40.0%	60.0%	100.0%
RegexpSingleline (8678) –	3.2%	5.8%	1.4%	2.6%	6.1%
RegexpSinglelineJava (588) –	43.2%	0.2%	7.7%	0.0%	43.5%
RightCurly (540) –	41.9%	62.8%	5.7%	28.0%	69.6%
SeparatorWrap (7) –	100.0%	14.3%	14.3%	28.6%	100.0%
SingleSpaceSeparator (22) –	81.8%	9.1%	95.5%	4.5%	100.0%
TrailingComment (552) –	71.9%	0.7%	29.9%	0.0%	75.7%
WhitespaceAfter (1053) –	96.5%	82.9%	30.8%	65.1%	98.6%
WhitespaceAround (1016) –	97.4%	79.9%	62.8%	42.9%	99.9%
	Styler	Checkstyle-IDEA	Naturalize	CodeBuff	All

**Fig. 6** Types of Checkstyle violation repaired per tool

violations of 21 rules. Surprisingly, NATURALIZE produced fixes for a higher number of violation types than CS-IDEA, even though it does not consider the Checkstyle ruleset of projects because of its different goals. CODEBUFF performed relatively well considering that it does not target Checkstyle violations as NATURALIZE. These facts suggest that our idea of employing a machine learning approach for repairing format checker violations is promising.

The reason for the high overall repairability of CS-IDEA (found in RQ #1) is that it outperformed the other tools in the five most frequent rules in our dataset: *RegexpSingleline*, *EmptyLineSeparator*, *LineLength*, *FileTabCharacter*, and *Indentation*. This explains why CS-IDEA is the leading tool in terms of repairability on the entire dataset but not on a sample of it, as shown in RQ #1. STYLER, on the other hand, had a perfect success rate in

rules that are not that frequent: *EmptyForIteratorPad*, *GenericWhitespace*, *ParenPad*, *RegexpMultiline*, and *SeparatorWrap*. Interestingly, CS-IDEA did not repair, at least not fully, any violation of two of these rules. This indicates that the tools are complementary to each other. Moreover, STYLER performed very well, with at least 80% of repaired violations, in the rules that are related to horizontal whitespace between two Java tokens, such as *MethodParamPad*, *NoWhitespaceAfter*, and *WhitespaceAround*. For developers, even if fixing these types of violations is easy, they may have dozens of them, which could be overwhelming. To that extent, automation is still valuable. Moreover, STYLER is able to repair these violation types for which one would not need to put engineering effort to write the repair code. Finally, we observed that all tools performed poorly on violations of the most frequent type in our dataset, i.e., *RegexpSingleline*.

**RQ #2: To what extent does STYLER repair different violation types, compared to other systems?**

STYLER and NATURALIZE repaired violations of a greater diversity of Checkstyle rules (24/25) than the other tools (CS-IDEA and CODEBUFF: 21). STYLER performed well for fixing violations related to horizontal whitespace between Java tokens. For some rules, STYLER fixed all violations while CS-IDEA did not fix any and, for other rules, CS-IDEA had a much higher reparability than STYLER, suggesting that they can be considered as complementary in practice. Finally, we confirmed that CS-IDEA outperformed the other tools on the five most frequent violation types in our dataset.

### 5.3 Unsuccessful repair cases (RQ #3)

STYLER repaired violations of 24/25 Checkstyle rules, but it did not perform well for some rules as shown in RQ #2. To understand in which cases STYLER does not successfully generate repairs, we manually analyzed violations of the Checkstyle rules for which STYLER repaired less than 50% violations. The analysis was ad-hoc, where, for each rule, both repaired and non-repaired violations were investigated so that patterns of non-repaired violations or their contexts could be identified. We present the cases of unsuccessful repair we found as follows.

STYLER encodes a violation according to the source code position returned by a format checker which is, in this case, Checkstyle. However, in some cases, this position is not where a fix should be applied. For instance, for a violation of the type *OneStatementPerLine*<sup>22</sup>, a line break should be added in the column 19 or 20 of the line 42, just after the first statement. However, Checkstyle returns column 31, which is the end of the second statement. In such a case, STYLER tried to repair the violation in an inappropriate location.

Several non-repaired violations were inside comments. For instance, we found lines of comments exceeding the maximum length of characters, therefore triggering violations of the type *LineLength*<sup>23</sup>. We also found tab characters inside comments, triggering *FileTabCharacter* violations<sup>24</sup>. These violations also happen with strings<sup>25</sup>. STYLER does not handle cases in which comments or strings should be modified. This is a limitation of

<sup>22</sup><https://kth.github.io/styler-experiments/#!/violation/Angel-ML-angel/339>

<sup>23</sup><https://kth.github.io/styler-experiments/#!/violation/opencb-opencga/173>

<sup>24</sup><https://kth.github.io/styler-experiments/#!/violation/actiontech-txle/13>

<sup>25</sup><https://kth.github.io/styler-experiments/#!/violation/apache-iotdb/463>

STYLER due to its tokenization. Comments and strings are tokenized as a single token, i.e., STYLER does not take into account the separation of words.

Moreover, we found several occurrences of a case in which STYLER repaired a given violation but then another existing one, which was not previously reported by Checkstyle, was triggered. This case of only one violation being reported when multiple ones exist in files is recurrent and happens with *FileTabCharacter* violations. We were not aware of that at the time we built the real violation dataset, but when a file contains more than one tab character, Checkstyle reports only the first instance of it. In some cases, STYLER repaired the first instance<sup>26</sup>, but the next one was then raised by Checkstyle. Even though STYLER repaired the originally reported violation, it was not counted as a repaired violation in our study. Note that one criterion to select files containing violations when building the dataset was the existence of a single violation in them (see Section 4.3). This was a decision we made to guarantee we could automatically check if a given violation was fixed. In such a case with *FileTabCharacter* violations, however, we could not check that precisely.

Finally, we observed that *RegexpSingleline* violations are the most frequent ones in our dataset and are poorly handled not only by STYLER, but all tools (see the last column of Fig. 6). When analyzing the violations related to this rule and other regex ones, we found out that many violations are not related to formatting. Some examples are violations related to missing, wrong, or duplicated license header<sup>27</sup> and the usage of specific patterns, such as a tag in javadoc, that are forbidden in some projects<sup>28</sup>. Since these violations are not about formatting, they are not in the targeted violation types of STYLER and other tools. However, the occurrence of these violations is very frequent in our dataset and, consequently, the reparability of the tools for such regex violations is impacted. For instance, our dataset contains 8,678 *RegexpSingleline* violations, and 8,102 (93%) of them are non-formatting violations. The overall reparability results about that rule, as presented in Fig. 6, are 3.2% for STYLER, 5.8% for CS-IDEA, 1.4% for NATURALIZE, and 2.6% for CODEBUFF. Adjusting the reparability results of the tools by considering only the 576 *RegexpSingleline* violations that are about formatting, we found out that STYLER, CS-IDEA, NATURALIZE, and CODEBUFF repaired 45.1%, 86.8%, 21%, 38.9% of *RegexpSingleline* violations, respectively.

**RQ #3: What are the cases in which STYLER fails to generate a correct repair?**

STYLER does not generate a correct repair when Checkstyle returns a source code position other than the one to be modified and when the violation is inside comments or strings. Moreover, STYLER produces repairs that make Checkstyle raise non-originally-reported violations. These cases in which STYLER failed to generate a correct repair relate to some limitations of STYLER, which can be further addressed in new studies. Finally, we found out that most of the violations of regex rules are not about formatting, which explains why the reparability of the tools is low for these rules.

<sup>26</sup><https://kth.github.io/styler-experiments/#!/violation/actiontech-txle/19>

<sup>27</sup><https://kth.github.io/styler-experiments/#!/violation/neo4j-neo4j/0>

<sup>28</sup><https://kth.github.io/styler-experiments/#!/violation/apache-usergrid/0>

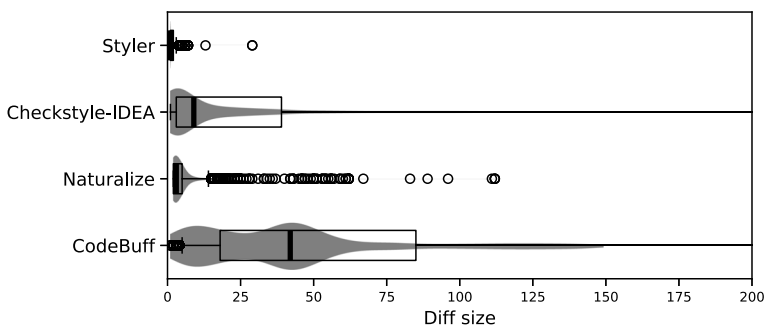
## 5.4 Size of the repairs (RQ #4)

One aspect of repair quality is the size of the diff between the source code with a formatting violation and the repaired source code. There might be different repairs for the same violation that pass all Checkstyle rules, but the one with the smaller diff size would be preferable for being the least disrupting for the developers. In the context of a pull request on GitHub, a smaller diff is usually considered as easier to review and merge (Dias 2020).

To answer RQ #4, we calculated the diff size, in number of lines, of the repairs generated by STYLER, CS-IDEA, NATURALIZE, and CODEBUFF. Figure 7 shows the distributions of diff size per tool. We observed that the distributions of the repairs generated by STYLER and NATURALIZE have the smallest medians, which are equal to one and three changed lines, respectively. Yet, they suffer from a few bad cases (the right-hand part of the distributions), mainly NATURALIZE. CS-IDEA and CODEBUFF produced larger repairs, with medians equal to nine and 42, respectively. In the worst cases, they produced several repairs with more than 200 changed lines, which can be seen by the fact that their 95th percentiles are not shown in Fig. 7. On the other hand, the 95th percentile of STYLER is three. We performed Wilcoxon rank-sum test to verify if the distributions of diff sizes obtained by STYLER and the other tools are significantly different from one another. We found  $p\text{-value} < 0.00001$  when testing STYLER with all the other tools. Considering  $\alpha = 0.05$ , we rejected the null hypothesis, which means that the distribution of STYLER is significantly different from the other ones.

### RQ #4: What is the size of the repairs generated by STYLER, compared to other systems?

The size of the repairs produced by STYLER is usually small. STYLER had the smallest median repair size of one changed line, followed by NATURALIZE, with a median size of three lines. CS-IDEA and CODEBUFF produced larger repairs. The ability to produce small diffs is an important property for code review and pull-request-based development, hence our results show that STYLER can be realistically used in a modern software development context.



**Fig. 7** Size of the repairs per tool. The two boxplot whiskers represent the 5th and the 95th percentiles

## 5.5 Performance (RQ #5)

To investigate if STYLER can be used in practice, we measured the time STYLER spent on the real violation dataset. Table 8 shows the minimum, median, average, and maximum time spent on the 104 projects, split over the different steps of the STYLER workflow. For training data generation, STYLER took at least 15 minutes and up to six hours, which is the maximum execution time allowed by our experimental setup (see Section 4.4.3). The median time for training data generation was 45 minutes. To tokenize the training data, STYLER took around two minutes on average, and a maximum of 14 minutes for training the models. The entire training process of STYLER (data generation + tokenization + model training) took around one hour and a half on average. This can be considered just fine since the training is meant to happen only when the coding conventions used in a project change (i.e., the Checkstyle ruleset file). After STYLER is trained for a given project, it takes on average two seconds to predict a repair, which is fast enough to be used in IDEs or in continuous integration environments.

### RQ #5: How fast is STYLER for learning and predicting formatting repairs?

On average, STYLER needs about one hour and a half for training and two seconds for predicting a repair. The training time is not an issue since it only happens when the Checkstyle ruleset file of a project changes. The prediction time relates to usability: our results show that STYLER can be used in IDEs or CI in a practical setting.

## 5.6 Technical analysis on STYLER (RQ #6)

At prediction time, STYLER used two trained LSTM models, each one based on a different training data generation protocol: *Styler<sub>random</sub>* and *Styler<sub>3grams</sub>*. In RQ #6, we investigated how the two protocols contribute to the final output of STYLER. We found out that STYLER fixed 967 violations exclusively with the *Styler<sub>random</sub>*-based model and 2,581 violations exclusively with the *Styler<sub>3grams</sub>*-based model. 7,460 violations were fixed with both models. This shows that the model based on the *Styler<sub>3grams</sub>* protocol is more effective than the model based on the *Styler<sub>random</sub>* one. In a real case scenario, one could consider using STYLER only with the *Styler<sub>3grams</sub>*-based model and still obtain fixes for 91% of what STYLER can repair. This would reduce the time for training STYLER.

**Table 8** Statistics on the performance of STYLER

	Training			Prediction
	Data generation	Tokenization	Models	Average Time
Step (Fig. 3) <sup>a</sup> :	A	B	C	E → I
Min	00:15:58	00:00:07	00:04:33	01.380 s/err
Med	00:45:19	00:01:36	00:07:07	02.133 s/err
Avg	01:23:05	00:01:53	00:07:13	02.267 s/err
Max	06:12:33	00:05:38	00:14:01	05.723 s/err

<sup>a</sup>The steps were executed in a computer containing a processor Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz and 125GiB system memory. For training the models, we used GPUs GeForce RTX 2080 Ti

**RQ #6: How do the two training data generation techniques of STYLER contribute to its reparability?**

The LSTM model based on the *Styler<sub>3grams</sub>* protocol is more effective than the model based on the *Styler<sub>random</sub>* protocol. Even though the latter exclusively contributes to the overall reparability of STYLER with 9% of the fixes, in a real case scenario, one could use only the former to reduce the time for training STYLER.

## 6 Discussion

We discuss, in this section, machine learning versus rule-based approaches, as well as the threats to the validity of our study.

### 6.1 Machine learning versus rule-based approaches

STYLER employs a machine-learning-based approach for repairing formatting convention violations. An alternative approach would be a rule-based one. In such a case, there would be at least one transformation to be applied in the code per format checker rule. However, the engineering of a transformation for every single linter rule is time-consuming. While this is costly, this might be even impractical for highly configurable linters such as Checkstyle, because the rule-based repair system would need to have different transformations for the same linter rule due to the configurable properties. On the contrary, a machine learning approach does not require costly human engineering. It is able to infer transformations for a diverse set of linter rules. Our experiments have validated this property in the context of formatting violations raised by Checkstyle. However, since our approach is far from being perfect and does not work well for certain rules, one avenue for future research is the development of a rule-based system for simple, non-highly configurable rules, to be complementary to STYLER.

### 6.2 Threats to validity

STYLER generates training data for learning how to repair violations based on the Checkstyle ruleset file contained in a given project. This means that STYLER assumes that all formatting rules contained in such a file are valid. In practice, however, developers might ignore the violations of certain rules. Our experiment does not take this scenario into account, thus we do not claim that all the fixes produced by STYLER are necessarily relevant for developers.

The real violation dataset contains Checkstyle violations mined from GitHub repositories. It is to be noted that it does not cover all existing Checkstyle formatting rules. Moreover, the dataset might not be representative of the real distribution of the 25 rules in the real world. Consequently, future research is needed to strengthen the validity of our study.

At the end of the violation collection process for building the dataset, we removed duplicate Java files according to file contents. However, there might still exist duplicate violations in our dataset. A file containing a violation might have changed, but that change might not be related to the line where the violation exists. Therefore, the same violation would exist in both versions of the file, but since the overall file contents are different, they are both kept in our dataset, which might create noise in it.

Another threat related to the creation of the dataset is that, when selecting violating files, we chose only the ones containing a single Checkstyle violation. We performed this selection so that we could accurately check if the violation was correctly repaired by the tools. Files containing more than one violation would make it hard to automatically check the correctness of repairs because once a violation is repaired, the location of the other ones in the file could be different. Therefore, our results are based on single-violation files, and future investigations on multiple-violation files are needed.

Finally, to compare the quality of the repairs produced by STYLER with the repairs produced by the other three tools, we measured the size in lines of the diff between the original and repaired program versions. However, the diff size is only one dimension for comparing the tools, which only approximates the developer's perception of formatting repairs. User studies, such as proposing formatting repairs to developers, are interesting future experiments to further investigate the practical value of this research.

## 7 Related work

STYLER aims to repair formatting violations raised by linters. Linters are a kind of automated static analysis tool. In this section, we first present works on the usage of static analysis tools. Then, we present systems that share similar goals with STYLER, which are systems that target linter violation repair and systems that fix source code formatting. Finally, we present works related to STYLER in terms of used technique, i.e., machine learning, for repairing compiler errors and behavioral bugs.

### 7.1 The usage of automated static analysis tools

Static analysis tools have been the subject of investigation in recent research. Zampetti et al. (2017) investigated their usage in 20 popular Java open source projects hosted on GitHub that use Travis CI to support CI activities. They first found out that the projects use seven static analysis tools—Checkstyle, FindBugs (Ayewah et al. 2008), PMD<sup>29</sup>, License Gradle Plugin<sup>30</sup>, Apache Rat<sup>31</sup>, Clirr<sup>32</sup>, and jDepend<sup>33</sup>—being Checkstyle the most used one. About the integration of static analysis tools in CI pipelines, they found out that build breakages due to those tools are mainly related to adherence to coding conventions, while breakages related to likely bugs or vulnerabilities occur less frequently. Zampetti et al. (2017) discuss that some tools are sometimes configured to just produce warnings without breaking the build, possibly because of the high number of false positives.

Vassallo et al. (2018) investigated the usage of static analysis tools from the perspective of the development context in which these tools are used. For that, they surveyed 42 developers and interviewed 11 industrial experts that integrate static analysis tools in their workflow. They found out that static analysis tools are used in three main development contexts, which are local environment, code review, and continuous integration. Moreover, they also found out that developers consider different warning types depending on the context, e.g., when performing code review they mainly look at style conventions and code redundancies.

<sup>29</sup><https://pmd.github.io/>, last access: 2020-07-13

<sup>30</sup><https://github.com/hierynomus/license-gradle-plugin>, last access: 2020-07-13

<sup>31</sup><https://creadur.apache.org/rat/>, last access: 2020-07-13

<sup>32</sup><http://www.mojohaus.org/clirr-maven-plugin/>, last access: 2020-07-13

<sup>33</sup><http://www.mojohaus.org/jdepend-maven-plugin/>, last access: 2020-07-13

Marcilio et al. (2019) focused on one specific static analysis tool: SonarQube<sup>34</sup>. Through an online survey with 18 developers from different organizations, they found out that most respondents agree that the issues reported by static analysis tools are relevant for improving the design and implementation of software.

## 7.2 Linter violation repair and code formatters

**Linter violation repair.** There are some tools to fix violations of rules checked by linters. Considering academic systems, there are Phoenix (Bavishi et al. 2019), which repairs violations of rules checked by FindBugs (Ayewah et al. 2008), and Getafix (Bader et al. 2019), which focuses on rules checked by Infer (Calcagno et al. 2015) and Error Prone (Aftandilian et al. 2012). These tools learn fix patterns by mining past human-written fixes for linter violations. Another tool is SpongeBugs (Marcilio et al. 2020), which repairs violations of rules checked by the two well-known static analyzers SonarJava and SpotBugs with fixed repair templates. STYLER shares with these tools the goal of generating patches for linter violations. However, while the mentioned tools focus on rules related to bugs and code smells, STYLER focuses on formatting. In addition, there is C-3PR (Carvalho et al. 2020), which does not generate patches itself but proposes fixes through pull request on GitHub generated by linter violation repair tools.

Beyond those academic systems, there are other tools that repair violations found by linters. Related to formatting rules, there is, for instance, ESLint, which is a linter for JavaScript, and it also includes automated solutions to repair violations raised by it.

**Code formatters.** A way to enforce formatting conventions lies in code formatters (also known as pretty-printers). In Section 4.2, we described NATURALIZE (Allamanis et al. 2014) and CODEBUFF (Parr and Vinju 2016). NATURALIZE recommends fixes for coding conventions related to naming and formatting in Java programs, and CODEBUFF infers formatting rules to any language given a grammar. Similar to the idea behind CODEBUFF, Reiss (2007) had previously experimented with different learning algorithms and feature set variations to learn the style of a given corpus so that it could be applied to arbitrary code. More recently, Markovtsev et al. (2019) presented STYLE-ANALYZER, which helps developers to fix code formatting during code reviews. STYLE-ANALYZER mines the formatting style of the git repository under analysis and expresses the found format patterns with compact human-readable rules. Then, it suggests style inconsistency fixes in the form of code review comments.

Beyond those academic systems, there are code formatters such as google-java-format<sup>35</sup>, which reformats source code according to the Google Java Style Guide<sup>36</sup>. However, these formatters are usually not configurable or require manual tweaking, which is a tedious process for developers. This is a problem because not all developers are ready to follow a unique convention style. STYLER, on the other hand, is generic and automatically captures the conventions used in a project to fix formatting violations.

Finally, there is the CHECKSTYLE-IDEA plugin for IntelliJ (CheckStyle-IDEA 2021), which we used to compare STYLER with. CHECKSTYLE-IDEA provides both real-time and on-demand scanning of Java files with Checkstyle from within IDEA. It also uses the

<sup>34</sup><https://www.sonarqube.org/>, last access: 2020-07-13

<sup>35</sup><https://github.com/google/google-java-format/>, last access: 2020-07-13

<sup>36</sup><http://checkstyle.sourceforge.net/reports/google-java-style-20170228.html>, last access: 2020-07-13

Checkstyle ruleset of projects to configure the formatter available in IntelliJ, making it possible to repair Checkstyle formatting violations. However, it is limited in repairing violations of a great number of Checkstyle rules as shown in RQ #2 and creates large repairs as shown in RQ #4.

### 7.3 Learning for repairing compiler errors and behavioral bugs

**Learning for repairing compiler errors.** There are related works in the area of automatic repair of compiler errors. In this case, the compiler syntax rules are the equivalent of the formatting rules. There, recurrent neural networks and token abstraction have been used to fix syntactic errors (Bhatia et al. 2018). In DeepFix, Gupta et al. (2017) use a language model for repairing syntactic compilation errors in C programs. Out of 6,971 erroneous C programs, DeepFix was able to completely repair 27% and partially repair 19% of the programs. Later, Ahmed et al. (2018) proposed TRACER, which outperformed DeepFix, repairing 44% of the programs. Santos et al. (2018) confirmed the efficiency of LSTM over n-grams and of token abstraction for single token compiling errors. These approaches do not target formatting violations, which is the target of STYLER.

**Learning for repairing behavioral bugs.** As for repairing compiler errors, there are also learning systems for repairing behavioral bugs, those that, for instance, break test cases. Tufano et al. (2018) investigated the feasibility of using Neural Machine Translation techniques for learning bug-fixing patches for real defects. They mined millions of buggy and patched program versions from the history of GitHub repositories and abstracted them to train an Encoder-Decoder model. The model was able to fix hundreds of unique buggy methods in the wild. Chen et al. (2019) proposed SequenceR, a program repair tool based on sequence-to-sequence learning focused on one-line fixes. In an experiment with Defects4J (Just et al. 2014), SequenceR was shown to be able to learn how to repair behavioral bugs by generating patches that pass all tests. STYLER and SequenceR share the same idea for formatting violation and bug encoding.

## 8 Conclusion

In this paper, we presented STYLER, which implements a novel approach to repair formatting violations raised by format checkers. STYLER creates a corpus of violations, learns from it, and predicts fixes for new violations, using machine learning. Currently, its implementation supports Checkstyle, a popular linter for Java programs. Our experimental results on 26,791 real Checkstyle violations showed that STYLER repairs real violations from a diverse set of Checkstyle rules and performs better for fixing violations related to horizontal whitespace between Java tokens than for fixing violations related to tabulations and line length. Moreover, STYLER produces smaller repairs than the compared systems, and its prediction time is low, which suggests that it can be used in development environments such as IDEs. Finally, we identified cases in which STYLER does not succeed to generate correct repairs, e.g., for Checkstyle violations inside comments or strings. These findings can guide improvements in STYLER and help researchers and developers to understand STYLER's limitations.

There are several interesting avenues for future research. First, improvements on the violation injection protocols for creating training data can be done to improve the representativeness of seeded formatting violations. This might increase the repairability of STYLER. Second, user studies can be conducted, where repairs predicted by STYLER are

proposed to developers through, for instance, pull requests on GitHub. This type of study would bring practical insights on the potential of STYLER. Third, STYLER could be integrated into development environments, such as IDEs and social coding sites, for supporting the mentioned user studies and possibly for developers to use STYLER. Fourth, other linters could be plugged in STYLER so it could be applicable on projects that use linters other than Checkstyle. Fifth, since STYLER does not work well for certain rules, the development of a rule-based system for simple, non-highly configurable rules, could be beneficial to complement STYLER. Finally, the overall idea behind STYLER could be tried out to repair other linter violations beyond purely formatting ones.

**Acknowledgments** We thank the anonymous reviewers for their valuable feedback that made us improve this paper. We also thank Terence Parr, Miltos Allamanis, Vadim Markovtsev, Hugo Mougard, Matias Martinez, and the ASSERT members who gave feedback on a draft version of this paper. Fernanda Madeiral would like to thank Thomas Durieux for his support on this work. This work was partially supported by the Swedish Foundation for Strategic Research under the TrustFull project, and by the Knut and Alice Wallenberg Foundation under the Wallenberg AI, Autonomous Systems, and Software Program (WASP). Some experiments were performed on resources provided by the Swedish National Infrastructure for Computing.

**Funding** Open access funding provided by Royal Institute of Technology.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

- Aftandilian E, Sauciu R, Priya S, Krishnan S (2012) Building Useful Program Analysis Tools Using an Extensible Java Compiler. In: Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '12), IEEE Computer Society, USA, pp 14–23 <https://doi.org/10.1109/SCAM.2012.28>
- Ahmed UZ, Kumar P, Karkare A, Kar P, Gulwani S (2018) Compilation Error Repair: For the Student Programs, From the Student Programs. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '18), Association for Computing Machinery, New York, NY, USA, pp 78–87, <https://doi.org/10.1145/3183377.3183383>
- Allamanis M, Barr ET, Bird C, Sutton C (2014) Learning Natural Coding Conventions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14), Association for Computing Machinery, New York, NY, USA, pp 281–293 <https://doi.org/10.1145/2635868.2635883>
- Ayewah N, Hovemeyer D, Morgenthaler JD, Penix J, Pugh W (2008) Using Static Analysis to Find Bugs. IEEE Software 25(5):22–29. <https://doi.org/10.1109/MS.2008.130>
- Bader J, Scott A, Pradel M, Chandra S (2019) Getafix: Learning to Fix Bugs Automatically. Proceedings of the ACM on Programming Languages 3(OOPSLA) <https://doi.org/10.1145/3360585>
- Bavishi R, Yoshida H, Prasad MR (2019) Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), Association for Computing Machinery, New York, NY, USA, pp 613–624, <https://doi.org/10.1145/3338906.3338952>
- Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '16), IEEE, pp 470–481 <https://doi.org/10.1109/SANER.2016.105>
- Beller M, Gousios G, Zaidman A (2017) Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17), IEEE Press, pp 356–367 <https://doi.org/10.1109/MSR.2017.62>

- Bhatia S, Kohli P, Singh R (2018) Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In: Proceedings of the 40th International Conference on Software Engineering (ICSE '18), Association for Computing Machinery, New York, NY, USA, pp 60–70 <https://doi.org/10.1145/3180155.3180219>
- Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, O'Hearn P, Papakonstantinou I, Purbrick J, Rodriguez D (2015) Moving fast with software verification. In: Havelund K, Holzmann G, Joshi R (eds) *NASA Formal Methods*, Springer International Publishing, Cham, pp 3–11
- Carvalho A, Luz W, Marcílio D, Bonifácio R, Pinto G, Canedo ED (2020) C-3PR: A Bot for Fixing Static Analysis Violations via Pull Requests. In: Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '20), IEEE, pp 161–171 <https://doi.org/10.1109/SANER48275.2020.9054842>
- CheckStyle-IDEA (2021) CheckStyle-IDEA. <https://plugins.jetbrains.com/plugin/1065-checkstyle-idea>, last access: 2020-07-13
- Chen Z, Komrusch S, Tufano M, Pouchet LN, Poshyvanyk D, Monperrus M (2019) SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* <https://doi.org/10.1109/TSE.2019.2940179>
- Dias H (2020) The anatomy of a perfect pull request: How does a good pull request look like. <https://medium.com/@hugooodias/the-anatomy-of-a-perfect-pull-request-567382bb6067>, last access: 2020-07-13
- Gupta R, Pal S, Kanade A, Shevade S (2017) DeepFix: Fixing Common C Language Errors by Deep Learning. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI '17), AAAI Press, pp 1345–1351
- Just R, Jalali D, Ernst MD (2014) Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14), Association for Computing Machinery, New York, NY, USA, pp 437–440 <https://doi.org/10.1145/2610384.2628055>
- Marcilio D, Bonifácio R, Monteiro E, Canedo E, Luz W, Pinto G (2019) 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), IEEE Press, pp 209–219 <https://doi.org/10.1109/ICPC.2019.00040>
- Marcilio D, Fúria CA, Bonifácio R, Pinto G (2020) SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings. *Journal of Systems and Software* 168:110671. <https://doi.org/10.1016/j.jss.2020.110671>. <https://www.sciencedirect.com/science/article/pii/S016412122030128X>
- Markovtsev V, Long W, Mougard H, Slavnov K, Bulychev E (2019) STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms. In: Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19), IEEE Press, pp 468–478 <https://doi.org/10.1109/MSR.2019.00073>
- Monperrus M (2018) Automatic Software Repair: A Bibliography. *ACM Computing Surveys* 51(1) <https://doi.org/10.1145/3105906>
- Parr T, Vinju J (2016) Towards a Universal Code Formatter through Machine Learning. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE '16), Association for Computing Machinery, New York, NY, USA, pp 137–151 <https://doi.org/10.1145/2997364.2997383>
- Prause CR, Jarke M (2015) Gamification for Enforcing Coding Conventions. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15), Association for Computing Machinery, New York, NY, USA, pp 649–660 <https://doi.org/10.1145/2786805.2786806>
- Reiss SP (2007) Automatic Code Stylizing. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07), Association for Computing Machinery, New York, NY, USA, pp 74–83. <https://doi.org/10.1145/1321631.1321645>
- Santos EA, Campbell JC, Patel D, Hindle A, Amaral JN (2018) Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors. In: Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '18), IEEE, pp 311–322 <https://doi.org/10.1109/SANER.2018.8330219>
- Santos RMD, Gerosa MA (2018) Impacts of Coding Practices on Readability. In: Proceedings of the 26th International Conference on Program Comprehension (ICPC '18), Association for Computing Machinery, New York, NY, USA, pp 277–285 <https://doi.org/10.1145/3196321.3196342>
- Spinellis D (2011) elyts edoC. *IEEE Software* 28(2):104. <https://doi.org/10.1109/MS.2011.31>
- Tufano M, Watson C, Bavota G, Di Penta M, White M, Poshyvanyk D (2018) An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), Association for Computing Machinery, New York, NY, USA, pp 832–837, <https://doi.org/10.1145/3238147.3240732>

- Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC (2018) Context Is King: The Developer Perspective on the Usage of Static Analysis Tools. In: Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '18), IEEE, pp 38–49 <https://doi.org/10.1109/SANER.2018.8330195>
- Yasunaga M, Liang P (2020) Graph-based Self-Supervised Program Repair from Diagnostic Feedback. In: International Conference on Machine Learning (ICML)
- Zampetti F, Scalabrino S, Oliveto R, Canfora G, Di Penta M (2017) How Open Source Projects use Static Code Analysis Tools in Continuous Integration Pipelines. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17), IEEE Press, pp 334–344 <https://doi.org/10.1109/MSR.2017.2>

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Benjamin Lorient** is a software engineer working in private sector. He recently received a Master's degree in Computer Science engineering from the Compigne University of Technology. His academic work is primarily focused in software engineering, particularly in automated code repair.



**Fernanda Madeiral** is a postdoctoral researcher at the KTH Royal Institute of Technology, Sweden. Her broad research goal is to help developers produce and maintain high-quality software systems. She is interested in software bugs and their fixes, linter violation, and source code legibility and readability.



**Martin Monperrus** is Professor of Software Technology at KTH Royal Institute of Technology. He received a Ph.D. from the University of Rennes, and a Master's degree from Compiègne University of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, program hardening and chaos engineering.

## Affiliations

**Benjamin Lorient<sup>1</sup> · Fernanda Madeiral<sup>2</sup>  · Martin Monperrus<sup>2</sup>**

Benjamin Lorient  
blorient97@gmail.com

Martin Monperrus  
martin.monperrus@csc.kth.se

<sup>1</sup> University of Technology of Compiègne, Compiègne, France

<sup>2</sup> KTH Royal Institute of Technology, Stockholm, Sweden