Applying probabilistic models to C++ code on an industrial scale

Andrey Shedko*†
a.shedko@samsung.com
Samsung Research Russia
Moscow, Russia

Ilya Palachev*
i.palachev@samsung.com
Samsung Research Russia
Moscow, Russia

Andrey Kvochko*
a.kvochko@samsung.com
Samsung Research Russia
Moscow, Russia

Aleksandr Semenov
alex.semenov@samsung.com
Samsung Research Russia
Moscow, Russia

Kwangwon Sun
kwangwon.sun@samsung.com
Samsung Research, Seoul, Korea

ABSTRACT

Machine learning approaches are widely applied to different research tasks of software engineering, but C/C++ code presents a challenge for these approaches because of its complex build system. However, C and C++ languages still remain two of the most popular programming languages, especially in industrial software, where a big amount of legacy code is still used. This fact prevents the application of recent advances in probabilistic modeling of source code to the C/C++ domain.

We demonstrate that it is possible to at least partially overcome these difficulties by the use of a simple token-based representation of C/C++ code that can be used as a possible replacement for more precise representations. Enriched token representation is verified at a large scale to ensure that its precision is good enough to learn rules from.

We consider two different tasks as an application of this representation: coding style detection and API usage anomaly detection. We apply simple probabilistic models to these tasks and demonstrate that even complex coding style rules and API usage patterns can be detected by the means of this representation.

This paper provides a vision of how different research ML-based methods for software engineering could be applied to the domain of C/C++ languages and show how they can be applied to the source code of a large software company like Samsung.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; **Software maintenance tools**.

KEYWORDS

code formatting, naming, C/C++, API rules, machine learning;

*These authors contributed equally
†Also with *National Research Nuclear University MEPhI*, Cybernetics.

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3391477

ACM Reference Format:
Andrey Shedko, Ilya Palachev, Andrey Kvochko, Aleksandr Semenov, and Kwangwon Sun. 2020. Applying probabilistic models to C++ code on an industrial scale. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3387940.3391477

1 INTRODUCTION

Recent trends in software engineering research show that machine learning methods can be successfully applied for different tasks: code completion [23], programming language machine translation [1], bug finding [21], mining of fix patterns [25], automatic repair [27] and many others. While the range of applications is quite wide, most publications consider only Java, Python, JavaScript and other languages for which the abstract syntax tree (AST) can be built in straightforward way. In case of C/C++, to build AST one needs to reproduce or capture the compilation of the project. Since different projects have different build scripts, it is impossible to build them all within a single simply described procedure. Moreover, it is very computationally expensive, because the whole compilation (with compiler optimizations and backend) should be done in order to keep the build successful. Because of that, it is hard to use traditional AST-based code representations to infer rules from a big corpus of code written in C and C++ languages.

In this paper, we propose to use a customized token representation for machine learning tasks on C/C++ code. While it is less precise and informative than AST-based representations, token representation is easily obtained from the source code. Our implementation is based on `LibFormat`, a tokenization library used by `clang-format`[1], a widely used source code formatter for C, C++, Objective-C, Objective-C++ and other languages.

To show its effectiveness, we apply our representation to two problems: coding style rule inference and API usage pattern mining. We verify with several methods that our statistical model allows us to detect formatting and naming rules on Tizen source code with high (more than 0.95) precision and recall. We have also mined over 40000 C/C++ repositories and discovered 22 API usage rules. When evaluated on Tizen OS, violations of these rules have acceptable false alarm rate and often point out potential defects. Most of these defects cannot be found by classic static analyzers, because they refer to specific API usage rules that are not known a priori.

[1] https://clang.llvm.org/docs/ClangFormat.html

These results demonstrate that the representation can be used for building high-quality probabilistic language models of C and C++ code. Low computational cost of the method allows it to be applied on a massive scale.

2 PROBLEM STATEMENT

Recent rise of machine learning caused its application in numerous domains of human knowledge, and software engineering is not an exception. According to Allamanis et al. [3], more and more papers on software engineering tasks employ machine learning in some form. Usually machine learning models need a large enough data set for the trained model to generalize well. In software engineering domain, this requirement turns into the ability to obtain a consistent code representation for various code repositories with reasonable performance.

Different approaches use different code representations for training data set: tokens [7], parse trees [20], abstract syntax trees [5] or even more complicated graph structures [4]. In most cases, these representations need at least the precise syntactic code analysis to be done. For languages such as C and C++, this presents a major challenge: in order to analyze the code, it is necessary to use proper header dependencies, run preprocessor with consistent macro expansions, specify correct compilation flags.

This problem is traditional for static analysis tools, which usually overcome it by recording AST information during the build. That requires capturing a repository's build info or modifying the build scripts. For example, `scan-build` tool[2] captures compilation commands during the build and `Bear` tool[3] uses `LD_PRELOAD`-based substitution. However, it is nearly impossible to build an arbitrary repository and therefore extract its AST, so this solution can not be applied to big code datasets. Some OS distributions introduce complex systems of reproducible builds (RPM, Debian, ebuild, *etc*), but they are usually only used for the packages within that OS, which may not be enough to train a high-quality language model. If the build system of a code repository is not deployed, the only thing that is available is its raw source code and possibly some build script that in general case cannot be used without special knowledge about its configuration.

To apply probabilistic models to C++ code, one can choose two of the following three conditions:

(1) the model is trained on a large corpus of C++ source code repositories,
(2) the code representation used for training is fully precise,
(3) the computational complexity of training is low enough.

If the first condition is omitted, it means that we fully build source code of some small number of repositories and the model can learn some knowledge specific for these repositories only. If the second one is omitted, we can analyze a big number of repositories, but at the cost that the representation is not precise and complete. The third condition cannot be omitted in the practical case, because building many repositories may be computationally infeasible.

To overcome this challenge, we propose to use a token representation where tokens are classified with some level of precision. While the token representation is less powerful than AST, it can be obtained for a wide range of repositories. To make the tool scalable, we use the `LibFormat` library[4] that is used by the popular `clang-format` tool[5]. In sections below we try to answer the following research questions.

RQ1 : Is `LibFormat` token classification reliable?
RQ2 : Is token representation suitable for building high quality probabilistic models? In particular, (**RQ2a**) can it be used to detect a code style? And (**RQ2b**) can it be used to mine API usage patterns from C/C++ source code?
RQ3 : Can a token-based representation be used on a massive scale?

2.1 Coding style and problem of legacy code

Coding style is a crucial concept in software engineering. Although it does not influence the semantics of code, it can affect code readability and maintainability. Consistent formatting is important for software engineering as it allows people to focus on the problem the code is trying to solve instead of thinking about where to place a brace or a newline. Properly formatted code is also more aesthetically pleasing than randomly structured code.

A number of automatic formatting tools exist (like `yapf`[6] for Python, `gofmt`[7] for Golang, `rustfmt`[8] for Rust, `clang-format` for C and C++) and many projects and communities have established style guidelines. But still many communinites do not adhere to these best practices, chosing either not to define a unified style, or not to strictly enforce it (e.g. by integrating automatic formatting into version control or code review system). This may become a problem in the future. Large projects without a formatting style become less maintainable.

For such projects, we propose to infer the dominant style and enforce it on the whole project, or at least on newly written code. This approach causes minimum changes to the code base and makes it look uniform.

In order to detect the dominant formatting style we suggest using a token representation and detect style elements (spaces, newlines, alignments) based on it. In further sections we will describe this approach and explain how it can solve the problem.

2.2 API usage patterns and language models

Application Programming Interface (API) is a term that covers various specifications and rules of how to interact with some program component (class, in object-oriented architecture). For example, such rules may specify what methods of dynamic library should be called in order to do some action properly. They may describe the order of such calls, different variants of usage, optional actions and so on.

Many widely used APIs suffer from the lack of documentation. Documentation can be incorrect, outdated (if the library was changed and the documentation was not updated after that) or may not exist at all. In any case, it is essential to learn the correct way to use API.

[2] https://clang-analyzer.llvm.org/scan-build.html
[3] https://github.com/rizsotto/Bear
[4] https://clang.llvm.org/docs/LibFormat.html
[5] https://clang.llvm.org/docs/ClangFormat.html
[6] https://github.com/google/yapf
[7] https://golang.org/cmd/gofmt/
[8] https://github.com/rust-lang/rustfmt

A number of different API pattern mining approaches are described in the literature. Some of them are even applied to programs written in C. Out of these methods we have chosen to infer documentation from the data, i.e. try to learn an API usage pattern from an existing code.

Language models of source code, that were first introduced by Hindle et al. [14], ascribe properties of a natural language to source code. These models can be used to find causal relationships in the code. Language models assign probabilities to all sequences of tokens in the code and may be used to find what token should be placed at the given context.

In the following sections we will describe how we used a language model to mine API usage patterns on massive C and C++ codebases.

3 THE TOKEN REPRESENTATION

Lexical analysis (or tokenization) is the process of converting a program text into a sequence of tokens. Each token is assigned a kind, such as identifier, keyword, separator, operator, literal *etc.*

The token representation can be ambiguous (token meanings can be too broad), imprecise (e.g. class name can be identified as a function name, and so on) and incomplete (the information presented about the code is incomplete). However, this representation can be obtained from code without compilation while providing useful information about tokens. Deeper information can be obtained from tokenized code by using simple heuristics.

In the following section we describe our approach regarding the tokenized code applied to two different applications: coding style detection and API usage pattern mining.

3.1 The `LibFormat` token structure

`LibFormat` is a part of `Clang`[9] infrastructure that is used as a base library for `clang-format`.

This library provides source code tokenization without access to compiler options and dependencies (i.e. header files that may be hard to obtain without deploying a complex build environment). `LibFormat` library is flexible enough to be used as a general tokenizer, and we found it sufficient for all our needs.

The output of `LibFormat` is a list of tokens with their kinds (identifier, C/C++ keyword, separator, operator, literal and many others, 475 in total), their contextual meaning (for example, an identifier can be a function or a variable declaration, a function call argument, *etc.*), spaces and tabs around the token, level of nesting, simple connections between tokens (e.g. allowing to jump across compound statements) and other information.

Special care is given to preprocessor directives. If the code can be preprocessed differently depending on the values of macros used, `LibFormat` provides an enumeration of all possible preprocessing variants. Only one of these variants is actually turned into AST during the compilation. Without the deployment of build environment it's impossible to know which one it is.

3.2 Filtered token representation

One of main drawbacks that differs token representation from tree (AST) and graph structures is the lack of interconnections between entities in code. While AST can represent a connection between distant nodes, in the token representation all tokens are placed linearly.

As mentioned above, `LibFormat` already provides functionality to navigate through tokens (like opening and closing paren). We propose to introduce these connections by filtering those tokens that are connected to each other according to a certain heuristic.

Since `LibFormat` allows one to detect function calls and language keywords, we decided to construct a token stream similar to the one described in the Wang et al. paper [28]. The idea is to only leave tokens corresponding to function calls (only identifiers of functions that are being called) and control flow expressions (like `if`, `else`, `while`, `for`, *etc.*). As `LibFormat` can find the closing `}` for an opening `{`, we represent them accordingly (as `<end_if>`, `<end_while>`, `<end_for>`).

4 CODE MODELS

We evaluate the token-based representation described above on two practical tasks formulated in problem statement, namely "coding style rule inference" and "API usage pattern mining". This section describes how the code is modeled in these tasks. Both models are simple, but surprisingly effective in considered applications.

4.1 Coding style rule inference

In this work, we consider the following two types of coding style rules:

(1) **formatting style rules** describe how the code is positioned: indentation, spacing, newlines placement and alignment. These rules are quite formalizable, but there are different ways to formalize them. For example, `clang-format` supports 105 different formatting rules[10], while `ReSharper` made by JetBrains has 662 formatting options that can be used with C++[11]. These rules may coincide or, in more complex cases, only have some intersection;
(2) **naming style rules** regulate how words within an identifier are separated (under_scores or CamelCase), and whether the first letter should be capital or not. We consider these two types of rules and do not support more complex naming rules that are also possible, such as the prohibition of abbreviations and acronyms.

We implemented the inference of dominant values for the following rules, which are based on (but not necessarily exactly correspond to) `clang-format` rules. These rules break down to two groups. *Simple* rules take boolean or integer value:

- `AccessModifierOffset`,
- `AlwaysBreakBeforeMultilineStrings`,
- `BraceWrapping` (with two subrules: `AfterFunction` and `AfterNamespace`),
- `IndentCaseLabels`,
- `IndentWidth`,
- `KeepEmptyLinesAtTheStartOfBlocks`,
- `SpaceAfterTemplateKeyword`,
- `UseTab`.

[9] https://clang.llvm.org/

[10] https://clang.llvm.org/docs/ClangFormatStyleOptions.html
[11] https://www.jetbrains.com/help/resharper/EditorConfig_Index.html

Complex rules are:

- `AlwaysBreakAfterReturnType`,
- `AlwaysBreakTemplateDeclarations`,
- `NamespaceIndentation`,
- `SpaceBeforeParens`.

While the empirical value of any simple rule can be estimated in a straightforward fashion by counting occurrences of different boolean or integer values, for complex rules it is not that simple. One needs to take into account different factors to make an estimation.

For example, the rule `SpaceBeforeParens` takes three possible values: "always", "never" and "only for control statements". Any parenthesis can be or not be a part of a control statement. Let us consider the number of parens in control statements that have (N_{C+}) and do not have (N_{C-}) a space before them, and the number of parens in other (non-control) statements that have (N_{O+}) and do not have (N_{O-}) a space before them.

With these numbers we infer the following for the value of `SpaceBeforeParens` rule:

- "always" — both N_{C+} and N_{O+} are high,
- "never" is most probable when both N_{C-} and N_{O-} are high,
- "only for control statements" is most probable when both N_{C+} and N_{O-} are high,
- None of the three possible values are probable when both N_{C-} and N_{O+} are high. Intuitively, this is the case where the space before a parenthesis is placed if and only if it is not a part of a control statement.

Probability for each of these values can be calculated as follows:

- $P(\text{“}always\text{”}) = \frac{N_{C+} \cdot N_{O+}}{S}$,
- $P(\text{“}never\text{”}) = \frac{N_{C-} \cdot N_{O-}}{S}$,
- $P(\text{“}control\text{”}) = \frac{N_{C+} \cdot N_{O-}}{S}$,
- $P(\text{“}other\text{”}) = \frac{N_{C-} \cdot N_{O+}}{S}$,

where S is the total number of parens. This gives

$$P(\text{“}always\text{”}) + P(\text{“}never\text{”}) + P(\text{“}control\text{”}) + P(\text{“}other\text{”}) = 1$$

Other complex rules are estimated the same way.

Naming rules are evaluated separately for classes, `enums`, `enum` values, functions (both class methods and non-class functions are considered in the same category), `typedefs` (`typedefs` of function pointers and other types are considered separately) and variables (constant variables are considered separately from non-constant ones). These categories are named as follows: `Class`, `Enum`, `EnumVal`, `Func`, `TypedefFunc`, `TypedefType`, `VarConst` and `Var` — eight categories in total.

For each of these 8 categories of identifiers, two naming rules are considered: whether the first character is a capital letter (e.g. `FirstFuncChar`) and what word separation is used: under_score or CamelCase (e.g. `FuncWordSeparation`). As a result, we consider $8 \cdot 2 = 16$ naming rules in total. All naming rules are estimated as simple rules.

4.2 API usage pattern mining

To show the effectiveness of our filtered token representation, we applied it to the task of API usage pattern mining. We used a very simple pattern mining algorithm.

Table 1: Experimental datasets

	Tizen	Github
Files	173,552	30,170,993
Lines total	65,708,893	11,733,552,183
LOC	46,447,703	8,358,177,464

We calculate the confidence of a candidate pattern $(x_1, ..., x_n)$ of length n as its conditional n-gram probability

$$P(x_n|x_1, x_2, ..., x_{n-1}) = c(x_1, ..., x_n)/c(x_1, ..., x_{n-1}),$$

where $x_1, ..., x_n$ is a sequence of tokens filtered as described in Section 3.2, and $c(x_1, ..., x_n)$ is the number of occurences (support) of this sequence in the dataset.

Then we filter out patterns with confidence ≤ 0.9, patterns that have over 100 violations, as well as patterns that have no violations at all. More formally, the set of candidate patterns C is given by

$$C = \{(x_1, ..., x_n) | P(x_n|x_1, ..., x_{n-1}) \in [0.9, 1) \text{ and } c(x_1, ..., x_{n-1}) - c(x_1, ..., x_n) \leq 100\}$$

We also found the following filter heuristics useful when inspecting the revealed patterns:

- there are no repeating tokens in the sequence (this often means that arbitrary number of calls is possible),
- there are no tokens in which the first character is a letter and others are non-letters (this helps to filter our autogenerated code),
- there are no tokens that end in a digit (to remove stack unwinding test code),
- the last token is not `<end_if>`, unless it is preceded by `exit` (actually, often patterns with closing `<end_if>` mean nothing),
- and some others.

After applying these filters, we rank the candidate patterns according to the number of occurences $c(x_1, ..., x_n)$ and examine top 100 of them manually to find potential defects among their violations. Results of manual inspection are presented in the evaluation section.

5 EVALUATION

Two datasets were used in our experiments: one containing the source code of all packages of the Tizen OS, and another one consisting of over 40000 top-rated C and C++ repositories downloaded from Github using GHTorrent [11] database. Identical files were removed in both datasets. Dataset statistics are presented in Table 1.

5.1 Verification of token representation

To answer **RQ1** we cross-verified the correctness of `LibFormat` token classification with our custom AST-based token classifier. Both `LibFormat` and AST classifiers were used to classify 3,105,820 identifiers in the Tizen dataset, and then the results were compared.

To get the values for precision and recall, we assume that AST checker provides the ground truth about the class of the identifier and calculate the quality of `LibFormat` implementation relative to it.

Table 2: Results of cross-verification

Token type	Precision	Recall	F1 Score
Func	0.9483	0.9730	0.9605
Class	0.9703	0.9960	0.9830
Var	0.9415	0.9737	0.9573
Enum	0.9913	0.9678	0.9794
TypedefFunc	0.8619	0.9823	0.9182
TypedefType	0.9609	0.9331	0.9468
EnumVal	0.9810	0.9990	0.9899
VarConst	0.9412	0.9712	0.9559

Table 3: Detected formatting rules on GitHub repositories

Rule name	Dominant value	Frequency
AlwaysBreakBeforeMultilineStrings	false	76.02%
AlwaysBreakTemplateDeclarations	MultiLine	47.38%
BraceWrappingAfterFunction	false	78.24%
BraceWrappingAfterNamespace	false	84.03%
IndentCaseLabels	false	73.33%
KeepEmptyLinesAtTheStartOfBlocks	false	99.96%
SpaceAfterTemplateKeyword	false	80.28%
UseTab	Always	73.50%

The result of evaluation is presented in Table 2. We found that LibFormat token classification has both high precision and recall, and with the lowest F_1 score at 0.9182 we are reasonably certain that LibFormat token classification is reliable, which answers **RQ1**.

5.2 Detecting the style of big code

The answer to **RQ2** consists of two parts: whether token-based representation can be used to infer proper values for stylistic rules from code (**RQ2a**), and whether it is suitable for mining useful API usage patterns (**RQ2b**). This section provides the first part of the answer.

To verify the quality of rule inference we use a technique we call **predictable check**, which consists of the following two steps:

(1) Format the copy of Tizen source code with some popular coding style (we considered Chromium, GNU, Google, LLVM, Mozilla, WebKit and Kernel coding styles),
(2) Run style detector on the formatted Tizen code and compare detected values of stylistic rules with expected ones.

Table 5 shows correct prediction rate averaged over the 7 defined styles. According to the table, the average detection rate for all rules is 0.95, which is acceptable for use on production code. It also indicates that the answer to **RQ2a** is positive.

Using our style detector, we have checked dominant coding style for selected 567 packages from the Tizen OS that were written by Samsung developers. Based on the results of the analysis, we are going to make suggestions about what would be the uniform corporate coding style that should be used in the future. In addition to this, we checked the coding style of 40,000 GitHub projects. Tables 3 and 4 show the result of that analysis. Frequency column corresponds to the percent of repositories in which the specified value is dominant.

5.3 Detecting API usage rules

This section gives an answer to **RQ2b** (whether token-based representation is suitable for mining useful API usage patterns), as well as **RQ3**.

To give an answer to **RQ3**, we train the model on the massive Github dataset consisting of 30 million files. The entire training procedure took 5 days, distributed across 5 machines and parallelized using `GNU Parallel` [26] utility to take advantage of 32 CPU cores on each of them. Evaluation of the model on the Tizen dataset only takes 12 minutes on a single machine.

The candidate set obtained as a result of this evaluation was used to answer **RQ2b**. We inspected violations of top-100 rules sorted by the number of occurrences in the training dataset and found 119 potential defects corresponding to 22 of those rules. We illustrate and describe in detail 4 of these 22 rules. A brief description of all of them is presented in Table 6.

```
skb_queue_tail(&priv->rx_queue, skb);
usb_anchor_urb(entry, &priv->anchored);
ret = usb_submit_urb(entry, GFP_KERNEL);
 usb_put_urb(entry) ;
if (ret) {
        skb_unlink(skb, &priv->rx_queue);
        usb_unanchor_urb(entry);
        goto err;
}
```

(a) Violation of rule (`usb_anchor_urb, usb_submit_urb, <if>`)

```
if (gst_caps_can_intersect (src_caps, allowed)) {
  ...
  codec_data_buffer = gst_buffer_new_and_alloc (2);
  gst_buffer_fill (codec_data_buffer, 0, codec_data, 2);
  gst_caps_set_simple (src_caps, "codec_data", GST_TYPE_BUFFER, codec_data_buffer, NULL);
  // No gst_buffer_unref here!
}
```

(b) Violation of rule (`gst_buffer_new_and_alloc, gst_buffer_fill, gst_caps_set_simple, gst_buffer_unref`)

```
if (dsp_wdt.fclk) {
    dsp_wdt.iclk = clk_get(NULL, "wdt3_ick");
    if ( !dsp_wdt.iclk ) {
        clk_put(dsp_wdt.fclk);
        dsp_wdt.fclk = NULL;
        ret = -EFAULT;
    }
} else
    ret = -EFAULT;
```

(c) Violation of rule (`clk_get, <if>, IS_ERR`)

```
// LCOV_EXCL_START
builder = g_variant_builder_new(G_VARIANT_TYPE("a{sv}"));
...
if (pin)
    g_variant_builder_add(builder, "{sv}", "pin", g_variant_new("s", pin));
params =  g_variant_new("(a{sv})", builder);  // Last call to builder
// Unref for builder should be called here
reply = asp_dbus_method_call_sync(ASP_DAEMON_SESSION_INTERFACE,
                    "ConfirmSession", params, &error);
if (error != NULL) {
    ...
}
g_variant_get(reply, "(i)", &ret);
g_variant_unref(reply);
```

(d) Violation of rule (`g_variant_new, <end_if>, g_variant_new, g_variant_builder_unref`)

Figure 1: Examples of detected rule violations

Table 4: Detected naming rules on GitHub repositories

Token kind	Dominant word separation	Frequency	Dominant first char	Frequency
Func	under_bar	56.24%	Lower	79.67%
Var	under_bar	61.26%	Lower	97.35%
VarConst	under_bar	72.04%	Lower	88.43%
Class	CamelCase	52.88%	Upper	57.16%
Enum	CamelCase	53.87%	Upper	63.65%
EnumVal	under_bar	82.32%	Upper	85.35%
TypedefType	under_bar	61.23%	Upper	50.97%
TypedefFunc	under_bar	64.25%	Upper	50.43%

Table 5: Result of predictable check for different styles

Rule	Detection rate
AccessModifierOffset	0.9994
AccessModifierOffsetIndentWidth	0.9994
AlwaysBreakAfterReturnType	0.7691
AlwaysBreakTemplateDeclarations	0.7507
BraceWrappingAfterFunction	0.9944
BraceWrappingAfterNamespace	0.9984
IndentCaseLabels	0.9861
IndentWidth	0.9958
NamespaceIndentation	0.9992
SpaceAfterTemplateKeyword	0.9740
SpaceBeforeParens	0.9345
UseTab	0.9986
Average for all rules	**0.9500**

(1) In Figure 1a the `entry` object is freed in `usb_put_urb` and then possibly freed again in `usb_unanchor_usb`[12].
(2) There are 15 examples in our dataset where `gst_buffer_unref` function is called to decrement the number of references to a buffer after allocating it with `gst_buffer_new_and_alloc`, initializing with `gst_buffer_fill` and passing to `gst_caps_set_simple`. Figure 1b shows the only example[13] where `gst_buffer_unref` is not called, which strongly indicates a memory leak.
(3) Figure 1c shows an example[14] of an incorrect check of a function's return value. In 3099 cases the return value of `clk_get` function is checked by calling `IS_ERR(return_value)` inside an `if` condition. However, in this case, `IS_ERR` function is not called, and the return value is checked directly.
(4) Figure 1d illustrates an example[15] of a missing `unref` of a variant parameter builder. After the variant is built or in case of an error the builder should be deallocated along with the result of the call. However, it is not done in the example, which may lead to a memory leak.

Even though we did not achieve the state-of-the-art results, considering the simplicity of the model, they still indicate that **RQ2b** can be answered positively. As a future direction of work, more complex models can be used to prune false positives and detect more kinds of defects.

[12]Code was taken from the file `drivers/net/wireless/rtl818x/rtl8187/dev.c` of repository `profile/wearable/platform/kernel/linux-3.18-exynos7270`. Hereinafter all code examples are from https://review.tizen.org/git/
[13]Code was taken from the file `gst/audioparsers/gstaacparse.c` of repository `platform/upstream/gst-plugins-good`
[14]Code was taken from the file `drivers/staging/tidspbridge/core/wdt.c` of repository `profile/wearable/platform/kernel/linux-3.4-exynos3250`
[15]Code was taken form the file `src/asp-client.c` of repository `platform/core/api/asp`

6 THREATS TO VALIDITY

This work is based on several assumptions which, if not met, may affect the validity of our results.

(1) We rely on our AST checker to give the ground truth about a class to which an identifier belongs. It is possible that this may not hold in some instances, for example, due to a bug in the checker's code, which would affect the numbers in tables 2 and 4. However, we have not noticed such cases.
(2) We do our best to check correctness of detected formatting and naming rules on the Tizen dataset, for which we have the necessary tools to compile and get ASTs of all 567 packages. Results of these checks are presented in tables 2 and 5. We assume these numbers do not change much when our tools are applied to a bigger Github dataset, however, we have no way of verifying this due to issues described in the problem statement section of this work.
(3) Even though the token representation works well for our chosen probabilistic models, it may not apply to other machine learning algorithms where the hierarchy and dependencies between code fragments are important. We note, however, that parse trees can be constructed from tokens without compilation and may work for such models, but such parsers too would need to go through a thorough verification procedure.

7 RELATED WORK

A lot of research has been done in the area of machine learning on source code in recent years. This section mentions works that are the most relevant to ours.

7.1 N-gram language models and token representation

The approach described in this paper uses the simplest configuration: token code representation and n-gram-based language model. These two concepts have been used for different applications of source code modeling:

- The probabilistic models of code based on token representation, as the simplest one, are used in various domains, such as learning coding conventions [2], migration [1], automatic syntax error correction [7], code completion [14], bug finding [22] and many others.

Table 6: Useful API patterns

Rule	Classification	Support	Violations	Potential defects
<if>, fprintf, exit, <end_if>	Dead code	4508	5	3
<if>, perror, exit, <end_if>	Dead code	1135	9	9
<if>, printf, exit, <end_if>	Dead code	1130	4	3
fprintf, strerror, exit, <end_if>	Dead code	638	6	5
com_err, exit, <end_if>	Dead code	392	10	1
strcmp, printf, exit, <end_if>	Dead code	139	1	1
<if>, usage, exit, <end_if>	Dead code	123	1	1
get_fs, set_fs, filp_open, <if>	Dead code	54	1	1
clk_get, <if>, IS_ERR	Incorrect return value check	3099	183	72
usb_anchor_urb, usb_submit_urb, <if>	Double free	318	12	4
platform_get_resource, devm_ioremap_resource, <if>	Memory leak	324	3	1
release_mem_region, pci_resource_start, pci_resource_len	Memory leak	122	6	6
gst_buffer_new_and_alloc, gst_buffer_fill, gst_caps_set_simple, gst_buffer_unref	Memory leak	15	1	1
g_source_set_callback, g_source_set_name, g_source_attach, g_source_unref	Memory leak	14	1	1
dev_err, <end_if>, device_create_file, <if>	Unchecked return value	114	1	1
rt2x00_eeprom_dbg, <end_if>, rt2x00_eeprom_read, <if>	Unchecked return value	107	1	1
set_fs, filp_open, <if>	Unchecked return value	55	1	1
exit, <end_if>, open, <if>	Unchecked return value	36	1	1
make_kuid, current_user_ns, <if>, uid_valid	Unchecked return value	35	1	1
devm_ioremap_nocache, resource_size, <if>	Null pointer dereference	66	2	2
foreach_in_list, as_variable, <if>	Null pointer dereference	47	2	2
contacts_list_first, <do>, contacts_list_get_current_record_p, <if>	Wrong resource selection	25	1	1

- Different modifications of n-gram source code models include n-grams enriched by nested cache [13] or some other additional information (semantic or otherwise) [19, 23], n-grams on program dependence graphs (PDG) [15], n-grams and CRF on AST paths [5], combination of n-grams with other models [10].

The token representation in this paper is based on `LibFormat` library and is verified with checks described in the paper. Also, we describe how to make a filtered token-based representation based on it. Two quite different tasks are completed based on the same representation: while the full token representation is used for mining formatting rules, the filtered one is used for mining API usage rules.

The paper uses the n-gram model to produce patterns, not to detect anomalies and make predictions, which are the traditional purposes of the n-gram language model. This helps to select most relevant API usage patterns, which then can be used to find anomalies in the code.

7.2 Automatic formatting

Automatic formatting, namely formatting based on dominant coding style, has been already tried in different research projects [2, 9, 24] and industrial software development tools:

- ReSharper detects the style of C# code[16],
- `uncrustify`[17] detects most appropriate values of its own style options,
- Lookout STYLE-ANALYZER [17] produces human-readable coding rules.

On the contrary, Parr and Vinju [20] predict style elements like newlines and spaces without producing explicit formatting rules. This is less practical, because the user can not see interpretable rules that are used to format the code.

To the best of knowledge, our tool first detects values for options of widely used `clang-format` tool. After values of its options are detected, it is possible to use them with `clang-format` to reformat the code. It is potentially possible to extend our tool to produce values of all `clang-format` options, so that to generate full formatting config that would be useful afterwards. Such config could be checked into the repository and used for pre-commit checks during the further development.

7.3 API specification mining and anomaly detection

Mining API specifications from existing code already has a long history [6]. A recent approach by Murali et al. [18] uses a Bayesian framework to learn probabilistic specifications from source code, `Nar-Miner` [8] mines negative association rules $A \Rightarrow \neg B$ from source code.

[16] https://blog.jetbrains.com/dotnet/2018/12/05/detection-code-styles-naming-resharper/

[17] http://uncrustify.sourceforge.net/

Gruska et al. [12] implement a lightweight language-independent parser for C language and search for anomalies on 6000 open-source Linux projects. Function models extracted by the parser are similar to our filtered token representation, enriched by the information about call arguments and their positions. However, a rigorous verification of the parser is missing, and, considering the complexity of C++ grammar, this can play an important role in the overall quality of the model.

`PR-Miner` [16] infers API rules by building frequent itemsets from their "preprocessed identifiers" code representation. However, their representation requires type information for each variable, which can only reliably be obtained during compilation.

`Bugram` [28] tool is similar to ours, but their model is inherently not scalable. Results are the best when the model is trained and evaluated on the same project. However, our investigation showed that when it is trained on a massive number of projects, most anomalies it produces become false positives, and "false positive pruning" procedure described in the paper can not be applied because the intersection of anomalies of different lengths collapses into an empty set.

8 CONCLUSION

The paper presents an attempt to apply language models to C and C++ languages, which are one of the most complicated targets for machine learning models, due to the complexity of its processing and obtaining comprehensive dataset for learning. It demonstrates that the simplest code representation, i.e. token-based one, can be used for scalable learning. By verifying this representation we show that the obtained dataset has enough quality and thus can produce adequate patterns.

We believe that in order to make an appropriate impact on businesses, future AI-based software engineering tools should be applicable to industrial languages such as C and C++. Current level of tools does not provide enough support for these programming languages, therefore efforts should be spent to make these tools applicable to the C and C++ domain. We see current work as a step in this direction.

ACKNOWLEDGMENTS

The authors of the paper would like to thank their colleagues from Samsung company, present as well as former ones, namely Evgeny Pavlov, Changhee Park, Youil Kim, Sergey Sobko, Elena Shapovalova, Andrey Visochan, Svetlana Onufrieva, Oleg Khokhlov, Anton Maslennikov, Slava Barinov and others who advised, proposed ideas and comments regarding this research.

REFERENCES

[1] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. *Using machine translation for converting Python 2 to Python 3 code.* Technical Report. PeerJ PrePrints.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 281–293.

[3] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices* 53, 4 (2018), 404–419.

[6] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.

[7] Sahil Bhatia and Rishabh Singh. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129* (2016).

[8] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 411–422.

[9] Filippo Corbo, Concettina Grosso, and Massimiliano Di Penta. 2007. Smart Formatter: Learning Coding Style from Existing Source Code. In *2007 IEEE International Conference on Software Maintenance*. 525–526. https://doi.org/10.1109/ICSM.2007.4362682

[10] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. Cacheca: A cache language model based code suggestion tool. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 705–708.

[11] Georgios Gousios. 2013. The GHTorent dataset and tool suite. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 233–236.

[12] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. 2010. Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/1831708.1831723

[13] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.

[14] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[15] Chun-Hung Hsiao, Michael Cafarella, and Satish Narayanasamy. 2014. Using web corpus statistics for program analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 49–65.

[16] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 306–315.

[17] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. 2019. STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 468–478.

[18] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian specification learning for finding API usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 151–162.

[19] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 532–542.

[20] Terence Parr and Jurgen Vinju. 2016. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 137–151.

[21] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[22] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 428–439.

[23] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.

[24] Steven P Reiss. 2007. Automatic code stylizing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 74–83.

[25] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2018. Learning quick fixes from code repositories. *arXiv preprint arXiv:1803.03806* (2018).

[26] Ole Tange. 2011. GNU Parallel: The Command-Line Power Tool. *;login:* 36 (2011).

[27] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720* (2019).

[28] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 708–719.