MSCCD: Grammar Pluggable Clone Detection Based on ANTLR Parser Generation

Wenqing Zhu
Nagoya University
Nagoya, Aichi, Japan
zhuwqing1995@ertl.jp

Norihiro Yoshida
Nagoya University
Nagoya, Aichi, Japan
yoshida@ertl.jp

Toshihiro Kamiya
Shimane University
Matsue, Shimane, Japan
kamiya@cis.shimane-u.ac.jp

Eunjong Choi
Kyoto Institute of Technology
Kyoto, Kyoto, Japan
echoi@kit.ac.jp

Hiroaki Takada
Nagoya University
Nagoya, Aichi, Japan
hiro@ertl.jp

ABSTRACT

For various reasons, programming languages continue to multiply and evolve. It has become necessary to have a multilingual clone detection tool that can easily expand supported programming languages and detect various code clones is needed. However, research on multilingual code clone detection has not received sufficient attention. In this study, we propose MSCCD (Multilingual Syntactic Code Clone Detector), a grammar pluggable code clone detection tool that uses a parser generator to generate a code block extractor for the target language. The extractor then extracts the semantic code blocks from a parse tree. MSCCD can detect Type-3 clones at various granularities. We evaluated MSCCD's language extensibility by applying MSCCD to 20 modern languages. Sixteen languages were perfectly supported, and the remaining four were provided with the same detection capabilities at the expense of execution time. We evaluated MSCCD's recall by using BigCloneEval and conducted a manual experiment to evaluate precision. MSCCD achieved equivalent detection performance equivalent to state-of-the-art tools.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

Code Clone, Parser Generator, Clone Detection, Syntactic Analysis, Programming Language

ACM Reference Format:
Wenqing Zhu, Norihiro Yoshida, Toshihiro Kamiya, Eunjong Choi, and Hiroaki Takada. 2022. MSCCD: Grammar Pluggable Clone Detection Based on ANTLR Parser Generation. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524610.3529161

ICPC 2022, May 21–22, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9298-3/22/05…$15.00
https://doi.org/10.1145/3524610.3529161

1 INTRODUCTION

Programming languages (hereinafter referred to as "languages") are advancing rapidly, and various languages are being developed and used for different purposes. Even within the same language, the syntax is frequently updated. For example, Typescript has been updated at least 26 times (see Figure 1), including minor updates, since its first release in 2012. This is a blessing for practitioners who are free to choose the latest language for their purposes.

Code clone detection [15, 29, 32], a successful application of program analysis, is required to deal with a wide variety of languages and grammar definitions by practitioners [7, 33]. Code clone researchers frequently receive requests for clone detection tools that support new languages and grammatical definitions in their industry-academia collaboration activities.

However, it is unrealistic to support a wide variety of languages while frequently adapting to the changes in grammatical definitions [33]. NiCAD [29] is a widely-used code clone detection tool that allows a user to specify the analysis method for each language. However, it is difficult for software developers who do not have much knowledge of program analysis to describe the analysis method. Semura et al. developed a token-based clone detection tool, namely CCFinderSW [33], which employs a lexical analysis mechanism to allow users to flexibly change the grammar of comments, identifier names, and keywords according to the target language. However, because CCFinderSW only supports lexical analysis changes, it cannot detect Type-3 clones [6] (see Section 3) that contain syntactic differences.

As a practical tool that supports a wide range of languages while flexibly responding to frequently changing grammar definitions, we propose MSCCD (**M**ultilingual **S**yntactic **C**ode **C**lone **D**etector), a Type-3 code clone detection tool that allows users to input ANTLR grammar definition files. **The ability to input the grammar definition files of ANTLR, a widely used parser generator, is a practical design choice for developing a code clone detection tool that can respond quickly to frequently changing grammars**. Because the grammars-v4 repository[1] of ANTLR grammars has more than 150 grammar definition files and over 6000 commits since 2012, MSCCD, which can input ANTLR grammar files, can handle frequent grammar changes.

Once a user provides an ANTLR grammar file and target programs that follow the grammar, MSCCD detects Type-3 clones from

[1] https://github.com/antlr/grammars-v4

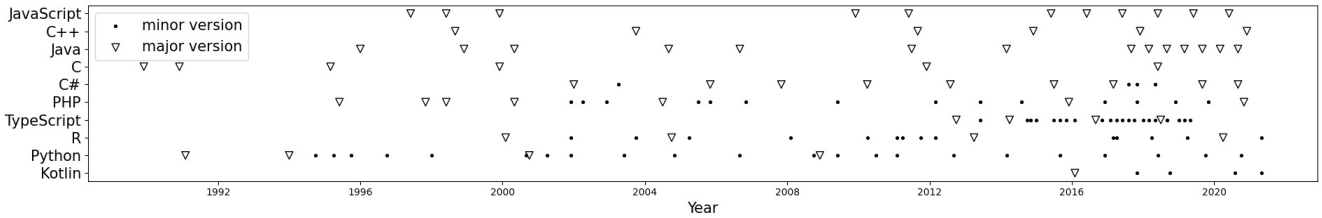


Figure 1: Release Log of Popular Languages

the target programs based on the grammar. By allowing the user to input the grammar definition file of ANTLR, MSCCD can be applied to programs written in many languages that have ANTLR grammar definition files. MSCCD first generates a parser that extracts the token bags (i.e., collections of elements with duplicates of keywords, identifiers, and literals) from the program according to the grammar definition file specified by the user. It then uses a parser to generate the token bags [32] and detects similar subsequences to identify Type-3 clones that contain syntactic differences.

We investigated the language extensibility of MSCCD to programs written in 21 widely used languages (see Table 2). We applied MSCCD to programs written in each of the 21 languages included in the Rosetta Code,[2] found that MSCCD can generate token bags for all the 20 languages whose grammar definition file exists in the grammars-v4 repository. In addition, we investigated the recall of MSCCD using a representative benchmark, BigCloneBench [35, 36] and found that the recall of MSCCD is comparable to that of SourcererCC [32], a state-of-the-art code clone detection tool (see Table 3). Then, we evaluated the precision of MSCCD for the source code included in BigCloneBench using the same procedure applied in extant SourcererCC research [32]. The results showed that MSCCD is slightly more precise than SourcererCC (see Table 3). Furthermore, MSCCD could complete the detection on a 100 MLOC source code collection in approximately 6 h (see Table 4).

The main contributions of this study are as follows:

- We provide a tool, MSCCD, which detects Type-3 clones from a target program according to its grammar when the target programs and an ANTLR grammar definition file are given. To the best of our knowledge, MSCCD is the first Type-3 clone detection tool that can be used with a grammar definition file.
- Evaluation experiments show that MSCCD supports most of the widely used languages and is competitive with the state-of-the-art Type-3 clone detection tools in terms of quantitative measures, such as precision, recall, and execution speed.
- MSCCD and its experimental data are available on the Internet[3]; this enables other researchers to reproduce the evaluation experiments.

The rest of this paper is organized as follows. Section 2 describes the motivations for this research, including language diversity and release frequency. Section 3 describes the important concepts and definitions. Section 4 introduces the proposed approach (i.e., token bag generation using a parse tree (PT)) and the implementation of MSCCD in detail. Section 5 describes various experiments conducted to answer the three research questions. Section 6 introduces the threats to validation. Section 7 presents the related work. Finally, Section 8 concludes the paper and discusses our future plans.

2 MOTIVATION

As programming languages evolve, code clone detection tools must keep pace. Figure 1 shows the release frequency of 10 popular languages (referring to the PopularitY of Programming Language Index ranking in September 2021[4]). A triangle indicates a major update, and a dot indicates a minor update[5]. Nearly all the languages are updated regularly, and several languages appeared within a decade of each other. Each release is likely to have introduced lexical or syntactic changes to the grammar, and code clone detection tools must be updated as these changes occur. In most cases, developers need to modify the source program to support these updates, making it challenging to keep most of the existing clone detection tools up to date.

Despite the large number of languages used in software development and the fact that these languages are regularly updated, the number of languages supported by most code clone detection tools is still limited. A survey paper on code clone detection research from 2013 to 2018 listed 13 tools [1], of which only one had a language extension mechanism. The other 12 tools only support a limited number of languages, including Java and *C/C++*. Research on code clone detection tends to be biased toward popular languages for which source code is plentiful and readily available. To the best of our knowledge, there is no large-scale benchmark that evaluates the recall of code clone detection in other languages, as BigCloneEval [36] only supports Java. Therefore, it is still difficult to evaluate the recall of code clone detection for other languages.

Among the existing tools, CCFinderSW [34] provides an extension mechanism to handle additional languages. This mechanism works by converting grammar definitions into regular expressions, targeting comments, string literals, and keywords. Regular expressions cannot express arbitrary context-free sentences; hence, CCFinderSW cannot support languages such as Lua. CCFinderSW only covers Type-2 clones, which is insufficient for many tasks. MSCCD can be used for many languages (probably more than 150) in the "grammars-v4" repository, and it can fully support new or

[2] http://rosettacode.org/wiki/Rosetta_Code
[3] https://doi.org/10.5281/zenodo.5886550

[4] https://pypl.github.io/PYPL.html
[5] For languages using the semantic versioning scheme (a system to manage version numbers semantically) [26], the major and minor versions are included. For other languages, larger versions equivalent to the major versions are included.

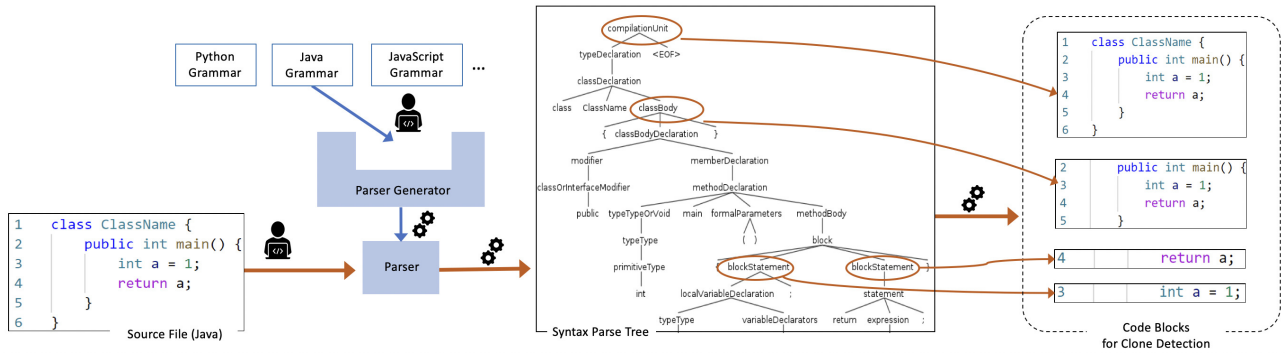


Figure 2: Multilingual Code Block Partition by Parse Tree

updated grammars by simply reusing the ANTLRv4 grammars, which are actively developed by the "grammars-v4" community, in a drop-in manner. Additionally, some tools emphasize easy expandability, such as SourcererCC [32]. However, we found that sometimes these tools do not obtain correct results. For example, we used SourcererCC's[6] block-level to tokenize a Java source file[7], but it failed to extract the correct blocks of some functions (lines 78–81) because the function parameters were split into several lines. Therefore, we are skeptical about the actual language extensibility of these tools. NiCAD [29], a widely-used code clone detection tool, allows the user to specify the analysis method for each language, but it is difficult for software developers who do not have much knowledge of program analysis to describe the analysis method.

Another possible solution to the rapidly changing language problem is to leverage an intermediate language such as the well-specified LLVM IR[8]. However, this solution cannot be used when the lexical and syntactic analyses of the program are required (e.g., syntactic clone detection [5, 13, 24]). Additionally, there are several languages for which no conversion tools to LLVM IR-type have been developed or for which a proper conversion is labor-intensive (e.g., dynamically typed languages, such as Python).

Based on these observations, this work aims to develop a tool that detects Type-3 clones from a target program according to the corresponding grammar, given the target programs and an ANTLR grammar definition files are given.

3 TERMINOLOGY

This paper uses the following definitions [28, 30]:

Token Bag: A bag (i.e., a collection of elements with duplicates) of keywords, identifiers, and literals.

Granularity Value: A non-negative integer indicating the level of the granularity of the code segment. Bigger granularity values correspond to the finer granularity.

Code Segment: A section of continuous lines of code which is defined by the quaternion (l, s, e, g), with the source file l, start line s, stop line e, and granularity value g.

Code Block: A code block is a code segment whose sentences are grouped by one grammar rule.

Composition: In this paper, a composition is defined as a code block corresponding to at least one grammar rule. Classes, condition statements, loop statements, or functions can be a composition. This item is mainly used to evaluate the ability of MSCCD to generate token bags in Section 5.

Clone Pair: A pair of similar code segments.

Clone Type: Code clones can be classified into four types:

- **Type-1 (T1)**: Identical code segments, except for the differences in white-space, layout, and comments.
- **Type-2 (T2)**: Identical code segments, except for the differences in identifier names and literal values, in addition to the T1 clone differences.
- **Type-3 (T3)**: Syntactically similar code segments that differ at the statement level. The segments have statements added, modified, and/or removed with respect to each other, in addition to the T1 and T2 clone differences.
- **Type-4 (T4)**: Syntactically dissimilar code segments that implement the same functionality.

4 PROPOSED TOOL

The following subsections introduce the main idea and implementation of MSCCD.

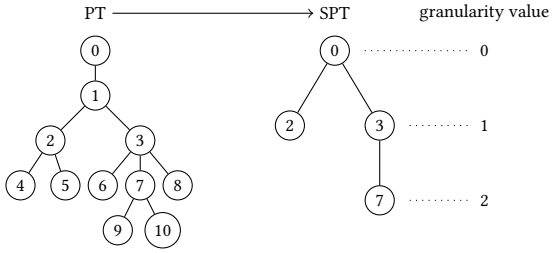
4.1 Main Idea: Code Block Partition by PT

Most code clone detection tools aim to not only detect clones between source files but also seek to partition them into code blocks [30]. This is a significant challenge for multilingual detection tools. Because the accurate division of code blocks requires syntax analysis and no syntax analyzer is suitable for multiple languages, replacing the syntax analyzer for the existing tools also requires source-code-level redevelopment. As shown in Figure 2, the main idea is that **every subtree in a parse tree (PT) presents a semantic code block**. A PT is an ordered tree representing the syntactic structure according to grammar. It is generated via syntax analysis, presenting how production is applied to replace non-terminals. Thus, each subtree in a PT represents the production of the grammar, wherein the root node represents the left side of the production, and all child nodes of the root node represent the right side of the production. Correspondingly, all leaf nodes from the subtree

[6] https://github.com/Mondego/SourcererCC
[7] https://github.com/tensorflow/Java/blob/daeb257/tensorflow-core/tensorflow-core-api/src/gen/annotations/org/tensorflow/op/DtypesOps.Java
[8] http://llvm.org

```
1  block           : '{' blockStatements? '}';
2  blockStatements: blockStatement+ ;
3  blockStatement :
       localVariableDeclarationStatement
4                  | classDeclaration
5                  | statement ;
```

Figure 3: A Part of the Java Grammar



*: The minimum tokens is set to 2.

Figure 4: Simplification of a Parse Tree

Table 1: The Number of Extracted Code Blocks

id	strategy	extracted code blocks
1	extract from PT	1 525 922
2	extract from simplified SPT	60 541
3	id 2 with keyword filter	35 556

*: The strategy 1 is not implemented in MSCCD.

present terminals, a token, an operator, or other lexical units. The source file can be divided into several blocks by handling these lexical units. Besides, a parser for the target language can be easily generated by using a parser generator.

Notably, the PT generated by a general-purpose parser is not suitable for code clone detection. On the one hand, a PT contains redundant nodes. For example, Figure 3 shows an example of the Java 8 grammar, which defines code blocks. For the derivation of the production in line 2, if the non-terminal *blockStatement* is matched only one time, the two subtrees (the root node of one is the non-terminal *blockStatements*, and the other is the non-terminal *blockStatement*) will contain the same lexical units. In other words, the two subtrees correspond to completely overlapping code blocks in the source file. This phenomenon is common in all languages. On the other hand, not all nodes represent a code block that can be regarded as a semantic code block. For example, many tiny subtrees may correspond to a part smaller than a statement. These parts are meaningless for code clone detection.

To reduce meaningless data, we propose simplifying the PT and generating a simplified PT (SPT). Algorithm 1 lists the steps required to simplify the PT. The input PT is traversed in the pre-order (line 2). The first step for each node, n, is to merge all child nodes containing the same lexical units as n (lines 3–5). The merged nodes are not visited afterward. The second step is to check if node n contains sufficient lexical units to meet the configured value (line

Algorithm 1: Parse Tree Simplification

Input: T is a PT, each tree node contains an attribute *size* representing its token number and an attribute *child* containing child nodes; *mSize* is the configured minimum size
Output: An SPT

```
1  Function ParseTreeSimplification(T, mSize):
2      foreach tree node n in pre-order traversal of T do
3          while n.length == n.child[0].length do
4              merge (n, n.child[0])
5          end while
6          if n.size < mSize then
7              foreach child node cn from n do
8                  delete (cn)
9              end foreach
10         end if
11     end foreach
12     return T
13 end
```

7). If node n is not large enough, all child nodes of n will be deleted to make n a leaf node (lines 8–10). The deleted nodes will not be visited in the traversal afterward. Figure 4 shows an overview of the simplification when setting *mSize* to two.

Each node in the SPT corresponds to a subtree in the original PT. Thus, it corresponds to a code block. The depth of a node in the SPT is defined as its granularity value. The smaller the depth value, the coarser the granularity. Furthermore, nodes having the same depth value are more likely to represent the precise composition of the language. For example, in Java, nodes having a depth value of one represent classes, and those having a value of four represent functions.

Additionally, languages use keywords to define semantic code blocks, such as classes, methods, and loops. These code blocks are also targets for code clone detection. This class of nodes in the SPT has a feature that has a child node corresponding to a subtree in the PT that only contains keywords. This feature allows for further filtering of nonsensical code blocks. We call it a keyword filter. It is discussed in Subsection 4.2.2.

To verify the effectiveness of PT simplification and the keyword filter, we conducted a preliminary experiment with the source code of Tensorflow-Java[9]. It contains 2,158 files with 563,059 lines of code. We calculated the number of code blocks extracted using this method under various conditions. The results are summarized in Table 1. When using the keyword filter, the number of target nodes was only approximately 2% of the original non-leaf nodes. Additionally, depending on the detection task's needs, the range of block size, granularity values, and keywords can also be configured to limit further the number of code blocks involved in clone detection.

The proposed method has three merits. First, **source files can be correctly divided into semantic blocks.** Because all subtrees in the PT correspond to a production in the grammar, the generated

[9] https://github.com/tensorflow/Java

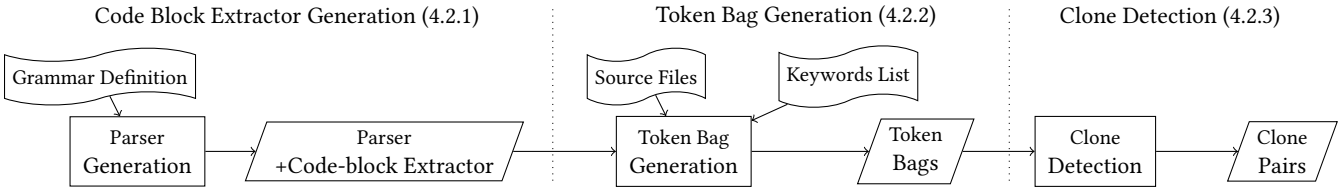


Figure 5: Overview of MSCCD

code blocks also contain the semantic information from that production. Additionally, via syntax analysis, the correctness of the code block partition can be ensured. Second, **Code blocks at various granularities can be generated by using this method.** In the PT, the node closer to the root node represents a greater granularity, such as a class or the whole file; the node farther from the root node represents a finer granularity, such as that of a statement. Thus, detectors can detect clones at multiple granularities or within a specific language component. Third, **high language extensibility can be ensured using a parser generator**. Theoretically, a parser generator can generate parsers for all context-free languages. The parsers generated by the same parser generator can be accessed using the same application programming interface (API). As a result, the supporting language can be changed without modifying the program.

4.2 Overview of MSCCD

The overview of the proposed code clone detection tool (i.e., MSCCD) is summarized in Figure 5. This depiction can be regarded in three phases: code block extractor generation (Section 4.2.1), token bag generation (Section 4.2.2), and clone detection (Section 4.2.3). First, MSCCD generates a code block extractor for the target language using a parser generator. Second, the input source code is converted into token bags. Finally, MSCCD detects the code clones between the token bags. Algorithm 2 lists the main steps of token bag generation in Section 4.2.2.

4.2.1 Code Block Extractor Generation. Here, MSCCD generates a code block extractor for the target language. For each language, MSCCD only needs a grammar definition file. Thus, MSCCD can change the supporting language or language version by changing only the grammar definition file, and there is no need to modify the program. Hence, MSCCD has excellent language extensibility. This only needs to be executed once for each language.

We chose ANLTR[10], which can create a parser for a language based on the grammar definition, including lexer and parser rules. For ANLTR, a language is defined by context-free grammar expressed using the extended Backus–Naur form (BNF). Therefore, grammar definition files for ANTLR can be easily created using the official grammar information. For languages that need unique treatments (e.g., the indents in Python), ANTLR allows programs to be embedded in the generated parser to solve the issue. Furthermore, there is a GitHub repository named "grammars-v4" [11], which is a collection of ANTLR grammars and handling programs with more than 150 languages. Users can easily obtain grammar definition files from this repository for a target language.

The code block extractor simplifies the PT generated by the parser using the algorithm introduced in Section 4.1 and identifies the code blocks.

Algorithm 2: Token Bag Generation

Input: T is a SPT generated by Algorithm 1. K is the keywords list.
Output: A Collection of token bags

```
1  Function TOKENBAGGENERATION ( T, K ):
2      TargetNodes,TokenBags = [];
3      if K == ∅ then
4          foreach TreeNode n in T do
5              TargetNodes.append(n);
6          end foreach
7      else
8          foreach TreeNode n in T do
9              if KeywordsFilter(n, K) == True then
10                 TargetNodes.append(n);
11             end if
12         end foreach
13     end if
14     foreach TreeNode tn in TargetNodes do
15         TokenBags.append( token bag created from tn )
16     end foreach
17     return TokenBags;
18 end
19 Function KeywordsFilter ( n, K ):
20     foreach TreeNode cn in n.childs do
21         flag = 0;
22         foreach Token t in cn do
23             if t.type == 'Keyword' then
24                 flag |= 1;
25             else
26                 flag |= 3;
27             end if
28         end foreach
29         if flag == 1 then
30             return True;
31         end if
32     end foreach
33     return False;
34 end
```

[10] https://www.antlr.org/
[11] https://github.com/antlr/grammars-v4

4.2.2 Token Bag Generation. The source files from the input project are partitioned and converted into token bags during this phase. The input software project, P, consists of several files, F: $P = \{F_0, F_1, ..., F_n\}$. A file, F, can be represented by a set of code blocks, SG. MSCCD allows overlap between code blocks because they have different granularity values, and overlaps are bound to occur. MSCCD transforms these code blocks into token bags [44], which are defined as sets of tokens.

For each source file, MSCCD first generates an SPT using Algorithm 1. Then MSCCD generates a token bag for each extracted code block. Algorithm 2 lists the steps of token bag generation. The input includes an SPT and a keyword list. All the sub-trees corresponding to code blocks is saved in the list $TargetNodes$. If the keywords filter is not activated, all the nodes from SPT will be set as target nodes which presents a code block (line 3-7). Only the nodes that passed the keywords filter are set as target nodes (line 8-14). At last, MSCCD generates token bags for each target node by accessing the corresponding sub-tree in the original PT (line 18).

The keyword filter can be activated by providing a non-empty keyword list. The input sub-tree can pass the keyword filter if at least one child only contains keywords (line 22-33). By using this filter, MSCCD retains more semantic code blocks to reduce the number of candidates. Additionally, this feature can be used to select specific parts of the language. For example, the user can detect code clones only between functions in Python by providing a keyword list that only contains the keyword "def".

4.2.3 Clone Detection. Code clones are detected by finding similar pairs from the generated token bags. To detect clones, we adopted the definition and algorithm proposed by SourcererCC, which has good recall and scalability for syntactic code clone detection [32]. It uses the ratio of the number of tokens shared by two token bags to the number of elements in the larger bags to present similarity. Two token bags, B_x and B_y, are judged as clones if the similarity is greater than or equal to the threshold, θ:

$$\frac{\left|B_x \cap B_y\right|}{MAX\left(\left|B_x\right|, \left|B_y\right|\right)} \geq \theta. \quad (1)$$

The main challenge in this part is organizing the token bags during detecting clones to obtain better performance. As explained in Section 4.2.2, a source file is presented by several token bags with several granularity values. Furthermore, there may be overlaps between token bags at different granularities. Theoretically, detecting clones between all the candidate pairs can achieve the highest recall. Correspondingly, this strategy has the highest operating overhead, significantly increasing execution time. Notably, the operating overhead increases due to the massive comparison times, and removing the overlapped clones from the report takes time. Overlapped clones are generated because the corresponding code segments may also be cloned, e.g., the corresponding sub-fragment of a T1 clone pair will still be a T1 code clone.

We chose to detect clones only between token bags with the same granularity value. In this strategy, when setting the biggest granularity value as g_{max} and the number of token bags in granularity value i as N_i, the time complexity of candidate comparison is reduced from $O\left(\left(\sum_{i=0}^{g_{max}} N_i\right)^2\right)$ to $O\left(\sum_{i=0}^{g_{max}} N_i^2\right)$. This is mainly because the functional consistency of code cloning makes it more likely to have the same granularity value, especially for inner-project clones. When the cloned code segment has a granularity value close to that of the original segment, there is a high probability that the clone will be detected due to the existence of the overlapped token bags. In other words, there will be a token bag containing the code segment at the corresponding granularity. Although the similarity of the code segments participating in the comparison is reduced, they can still be detected if higher than the threshold. Additionally, this method can be easily parallelized by using multiple processes to compare token bags in each granularity value.

For the overlapped clones, we choose only to report those of the smallest granularity value. The remaining overlapping clones will be filtered out after detection.

5 EVALUATION

To evaluate the performance of MSCCD, we conducted experiments to answer the following research questions:

- **RQ1**: How many languages can MSCCD support?
- **RQ2**: What is the precision/recall and the scalability of MSCCD code clone detection?
- **RQ3**: Are the code clones detected by MSCCD for each language appropriate for the purpose of software maintenance?

5.1 RQ1: Language Extensibility

Experimental Design: The proposed method includes an optional keyword filter. When executing MSCCD without using the keyword filter, it can theoretically support all languages with ANTLRv4 grammar files. However, this strategy increases the execution time because of a higher overlap of candidates. On the other hand, some code blocks might be filtered out incorrectly when using the keyword filter, thus reducing recall. This experiment evaluates the number of compositions MSCCD supports with or without using the keyword filter.

We determined four items for each language: class, function, branch, and loop. These items are the most basic and standard parts of languages. For each item, all corresponding grammar is included within the scope of the test targets. For example, all loop items, including but not limited to the while loop, for loop, and do-while loop, are test objects. We regard function items as the most relevant to show the extensibility of the target language among the four items. The other three items reflect whether MSCCD can detect code clones at more granularities. However, it should be noted that MSCCD does not merely support these four items. The other language components can also be accurately extracted depending on the grammar. If all corresponding code blocks are accurately extracted, including the corresponding code block, lacking any irrelevant code segments, the item is passed (circle mark; ●). It is regarded as a failure if a target code block fails to be generated correctly without a syntax analysis error.

Target Language and Input Data: We evaluated the most popular 21 languages according to the PYPL ranking. For each language, we randomly selected five files with more than ten lines from the Rosetta Code.[12]. Rosetta Code is a site that collects solutions to more than 1,138 tasks in various languages. It adopts basic methods

[12] http://www.rosettacode.org/wiki/Rosetta_Code

Table 2: Language Extensibility to the top 21 most popular languages in PYPL

Lang.	F.	Cl.	Cd.	Loop	Lang.	F.	Cl.	Cd.	Loop
Python	●●	●●	●●	●●	Kotlin	●●	●●	●●	●●
Java	●●	●●	●●	●●	Matlab	◖◖	☼	◖◖	◖◖
JavaScript	●●	☼	●●	●●	Go	●●	☼	●●	●●
C#	●●	●●	●●	●●	VBA	●●	☼	●●	●●
PHP	●●	●●	●●	●●	Rust	●●	●●	●●	●●
C	●●	☼	●●	●●	Ruby	◖⚹	◖⚹	◖◖	◖◖
C++	●●	●⚹	●●	●●	Scala	◖◖	◖◖	◖◖	◖◖
R	◖◖	☼	◖◖	◖◖	Ada	⌀	⌀	⌀	⌀
TypeScript	●●	●●	●●	●●	VB	●●	☼	●●	●●
Swift	●●	●●	●●	●●	Dart	●●	●⚹	●●	●●
Objective-C	●⚹	☼	●●	●●					

1: ●: Positive ⚹: Negative ☼: Not applicable ◖ : Positive, compilation error exists ⌀: Grammar definition does not exist
2: Left side: result without keyword filter. Right side: result with keyword filter.
3: Lang.: Language. F.: Function, Cl.: Class, Cd.: Condition.

and does not use a development framework; hence, the four items will appear in the code more commonly. If the test targets are randomly selected from open source software, the chosen targets are sometimes unable to obtain some items because these compositions may not be used in the source program.

Tool Configuration: In this experiment, we set the minimum tokens as two, which can fully expose the language component extraction ability of MSCCD. Moreover, the complete keyword list (the keyword list which contains all the keywords according to official information) is provided.

Result: Table 2 lists the experiment results. Among the 21 target languages, 20 are available in MSCCD. Moreover, all the components of these 20 languages can be supported when not using the keyword filter. Among the 20 languages, functions of 18 are successfully extracted except for Objective-C and Ruby. We manually checked the productions of functions in Ruby's grammar. We found that the root node of all the corresponded subtrees of the function composition can not contain a child node that only contains leaf nodes of keywords. Therefore, these subtrees in Ruby are filtered out by the keywords filter. For the same reason, only a part of the functions can be supported in Objective-C. All languages support the condition of the other three items, and the loop is supported by 19, except for Go. The reason for failing to extract the loop in Go is the same as that of the functions in Objective-C. For class, 11 passed the experiment among the 13 languages. The overall experiment results indicate that MSCCD has relatively high language extensibility.

Notably, the experiments of four languages (left circle mark; ◖) had syntax analysis errors and failed to generate a correct PT for some source files. Because the purpose of this experiment was to evaluate how many languages can be supported when the PT is generated, we manually checked the grammar of these languages to check the fact that these compositions can be extracted or not.

Comparison with CCFinderSW: Table 2 implies that MSCCD can be used for many languages in the "grammars-v4" repository. With state-of-the-art tools, CCFinderSW [34] has a close level of language extensibility. However, CCFinderSW's approach cannot support some languages e.g., Lua, making its language extensibility lower than that of MSCCD. One of the necessary steps for CCFinderSW to support a language is to convert the grammar rules into regular expressions. This conversion is not always possible since regular grammar is a subset of ANTLR's context-free grammar. Since MSCCD performs the syntactic analysis using ANTLR, such an issue does not arise. In addition, MSCCD can fully support new or updated grammars by simply reusing the ANTLRv4 grammars in a drop-in manner.

> The answer to RQ1: When using the keyword filter, MSCCD can provide function-level support for 18 of 20 available languages. Furthermore, when the keyword filter is inactivated, MSCCD can support all 20 tested languages.

5.2 RQ2: General Performance

We believe that because the similarity calculation results were the same, the detection performances of **MSCCD** remain unchanged regardless of the language. The experiments were conducted using Java. To compare the existing syntactic code clone detection tools, we followed the experimental methods published in SourcererCC [32] and used the framework and dataset provided therein. For all the experiments discussed in this subsection, we configured MSCCD for a minimum clone size of 50 tokens and a similarity threshold of 70%. A complete list of keywords is provided. The evaluation results of other compared tools were taken from the published work [41].

Recall: We measured the recall of MSCCD by using Big-CloneEval [36], which provides clone detection tool evaluations based on BigCloneBench [35]. BigCloneEval reports the recall of each type of code clone, which is used to measure the detection ability of syntactic code clones. In BigCloneEval, T3 clones were divided into Very-Strongly Type-3 (VST3), Strongly Type-3 (ST3), Moderately Type-3 (MT3), and Weakly Type-3 (WT3) according to the similarity [36]. We set the clone matcher to a minimum size of six lines and 50 tokens for comparing the existing results [41].

Table 3 shows the result of recall measurements. Similar to these tools, MSCCD also has a near-perfect recall on T1 and T2 clones. For the recall of the three T3 clones, MSCCD ranks third. Notably, the comparison with SourcererCC is interesting. The reason for this is explained as follows. The two tools use basically the same similarity calculation method and clone detection algorithm. In theory, the recall based on BigCloneBench should be the same. However, MSCCD recall on ST3 and MT3 is higher than that of SourcererCC. We believe that this is because MSCCD creates token bags in multi-granularity. When performing a complete detection, the same code segment is inspected at multiple granularities, improving the detection ability. To prove this argument, we set a test group for MSCCD containing only reported clones in granularity values 0 and 4 (file-level and function-level in Java). In this group, the recall of ST3 and MT3 dropped slightly and was closer to that of SourcererCC. The detection results in granularity values 0 and 4 only account for approximately 44% of the total. MSCCD reported a large number of clones at other granularities.

Precision: We measured the precision of MSCCD using the same random sample test as [32]. We randomly selected 400 clone pairs detected by MSCCD in the BigCloneEval experiment and equally distributed them to five judges to determine the correctness of each

Table 3: Recall and Precision Measurements

Tool	Recall					Precision
	T1	T2	VST3	ST3	MT3	
MSCCD	100	98	93	63	7	92
MSCCD*	100	98	93	61	6	91
SourcererCC	100	98	93	61	5	83
CCAligner	100	99	97	70	10	80
CCFinderX	100	93	62	15	1	72
Deckard	60	58	62	31	12	60
NiCad	100	100	100	95	1	56
iClones	100	82	82	24	0	91

*: only contains clone pairs in granularity value 0 and 4

Table 4: Execution Time

LOC	1K	10K	100K	1M	10M	100M
Time	1 sec	4 sec	17 sec	3 min 13 sec	1 hr 14 min 33 sec	6 hr 6 min 52 sec

clone pair. The clone pairs were marked as "unknown" for judging complex situations. To compare the state-of-the-art tools, we set two test groups. One group contained pairs selected from all the results. This group represents the average precision of MSCCD. The other group only contained reported pairs in granularity values 0 and 4. This group is more convincing to compare with the other tools because the granularity is closer to traditional tools.

Table 3 lists the results. For the first group containing results for all the granularities, the precision of MSCCD was calculated from the resulting 368 pairs of true positives and 32 pairs of false positives (including eight pairs marked as difficult to determine). MSCCD had the precision at 92%. For the group that only contains results in granularity values 0 and 4, MSCCD also had a near precision of 91%.

Execution Time: To evaluate the performance of different data sizes, we generated test files by randomly selecting files from IJaDataset[12]. We used the Linux command “wc” to measure lines of codes. The experiments were executed on a quad-core CPU, and the maximum heap memory size of the Java Virtual Machine was set to 12GB. The time to generate the parser and code block extractor was less than 5 s and only needed to be executed once. We also set two test groups to measure execution time when the keyword filter is activated or when it is not. The result is listed in Table 4. MSCCD has good scalability and can complete the 100-MLOC-level clone detection task in slightly more than 6 hr.

The answer to RQ2: MSCCD has a level of recall and precision equivalent to state-of-the-art tools and can complete detection on repositories of up to 100MLOC.

5.3 RQ3: Language Features of the Detected Clones

While some code fragments of code clones correspond to the logic, others are difficult to use for maintenance, such as a sequence of import or constant declarations. In this section, we present an experimental evaluation of whether MSCCD could detect code clones

Table 5: Overview of Detections in 9 languages

Language	MLoc	Tokens (M)	Token Bags (K)	Detected Clone mt = 20	Detected Clone mt = 50	Keywords Ratio
C	21.09	48.80	3405.84	10191	7148	15.44%
Java	4.62	13.20	391.61	504966	96004	37.68%
C++	3.14	9.22	126.18	55317	16823	47.89%
C#	2.10	4.35	117.98	100178	22333	62.40%
Kotlin	2.33	7.58	169.51	31805	16554	39.53%
Swift	0.28	0.65	18.82	59672	20121	49.94%
JavaScript	0.95	2.57	39.32	22230	10439	14.27%
Rust	1.65	3.91	90.90	6419	2040	38.48%
Go	7.42	22.20	35.52	9342416	1019142	10.30%

mt: min token. (M): mega. (K): kilo.

Table 6: Frequently Occurring Compositions in Figure 6

Lang.	GV	Comp.	Lang.	GV	Comp.	Lang.	GV	Comp.
Java	4	function	Swift	2	class	Go	1	function
C++	7	branch, loop		5	function		4	branch, loop
C#	6	function		8	branch	JavaScript	2	function
Kotlin	1	class	Rust	2	function	GV: Granularity Value		
	3	function		4	branch	Comp.: Composition		

of code fragments that represent logic in a wide range of minimum token numbers, which is a code clone detection parameter.

In the experiment, we selected five repositories from GitHub for nine target languages by order of stars. We believe that the high-star repositories are more representative. MSCCD was executed twice for each repository when configuring the minimum token number to 20 and 50. Table 5 lists an overview of the experiment, including size, number of extracted token bags, number of clones detected, and the ratio of keywords.

The language compositions that can be described at each granularity are different depending on the grammar. Therefore, we investigated how the granularity of the generated token bag and the detected clone changes when the min token is changed for each language. We drew the distribution of all the extracted token bags and all the token bags that were detected as clones along with each granularity value into a line graph (Figure 6). The x-axis of each graph represents the granularity value, and the y-axis represents the token bags and the reported clones corresponding to that granularity.

The first point of concern is the number and position of the heap in the polyline representing the clone (blue polylines). The heap in the blue polyline indicates that many clones were gathered at this granularity. Table 6 lists frequently occurring compositions at the granularity where there is a heap in Figure 6. We believe that function-level clones are more efficient for software maintenance. The clones in branches and loops are usually error handling, exception capture, and high-frequency code (such as traversal), which are relatively insignificant for software maintenance.

When the minimum number of tokens was changed from 50 to 20, some languages did not change the heap position very much, while others changed significantly. The small clones detected are often part of the larger clones in the languages whose heap did not change. Many small clones exist in the language whose heap

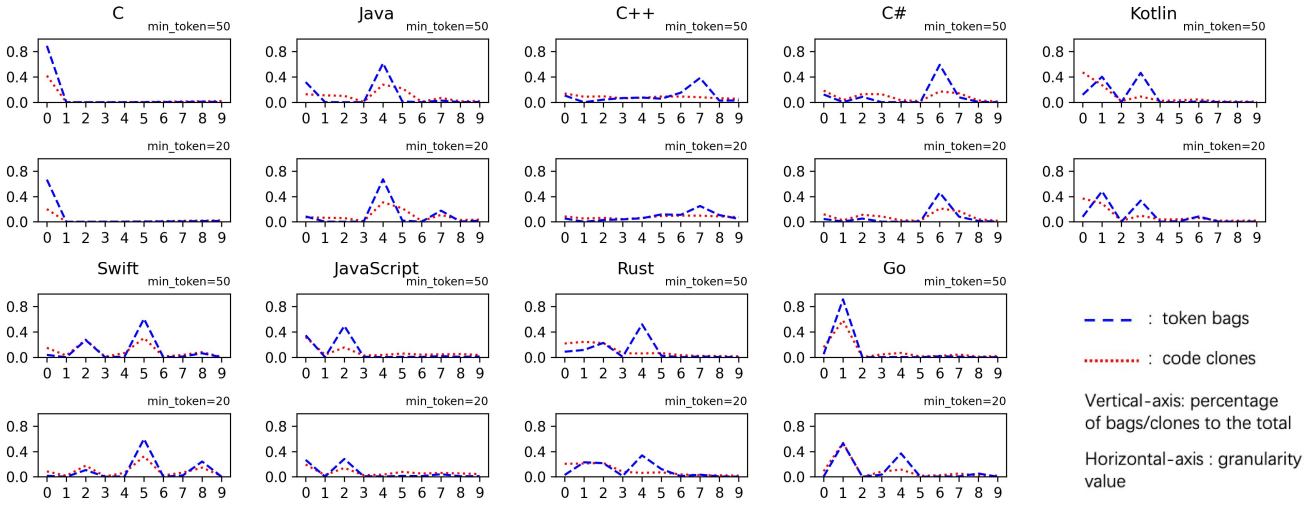


Figure 6: Distributions of Token Bags and Clones

changed that are not part of the large clones. So many tiny clones were reported. For example, in Go, the position of the clone curve heap changed significantly when the minimum number of tokens was 20. It occurred because Go's error handling generates an abundance of clones between branches. By contrast, for languages such as Java and JavaScript, the position of the heap does not change, but the height drops slightly. It showed few clones with smaller granularity values in this language, and most were produced in larger units. Tiny clones may reduce the effectiveness of the result set for maintenance but may be adequate for other tasks. Detection tools can be configured according to requirements.

Unlike other languages, the code clone of the C language is gathered at the granularity value of 0 whenever the minimum number of tokens is set to 20 or 50. However, this is not a feature of C language syntax but is related to application scenarios. The experimental object of the C language in this experiment includes the source code of the Linux Kernel. Many file-level clones exist because they need only individual replacements to adapt to different devices. The Linux Kernel project is significantly larger than the other tested projects, so the characteristics of the project overshadow the C language's characteristics.

The answer to RQ3: For most of the languages, MSCCD detected code clones such as functions, branches, and loops, which would be of interest for maintenance tasks when the parameter min token ranged from 20 to 50. For some languages, different min tokens may result in different kinds of clones being detected, which might be tuned by monitoring granularity.

6 THREATS TO VALIDITY

One internal threat is that we did not report the precision of different languages. We designed the experiments in Section 5.2 based on the following assumptions: Because MSCCD converts code blocks written in context-free languages into token bags, there will be no significant difference in the precision between different languages. The result shows that MSCCD has the same precision as clone detection tools for a specific language because MSCCD adopts the same detection method as the latest Type-3 clone detection tool (i.e., SourcererCC) after converting code blocks to token bags. Besides, MSCCD does not tune for any particular language. Making benchmarks by ourselves for this study may lack appropriateness and objectivity, so we abandoned the solution of reporting accuracy for each language separately and instead used the classical benchmark BigCloneBench for indirect evaluation.

We also did not report the recall of other languages. Firstly, we believe that the performance of the detection method is consistent in most programming languages. Secondly, there is a lack of recall evaluation benchmarks (e.g., BigCloneBench [35]), making it impossible to provide recall for all supported languages. Besides, BigCloneBench only contains code clones at the function level. That means the recall of clones with below granularities is not measured. As mentioned in Section 5, the number of clones reported at granularity values 0 and 4 is only about 44% of the total. From this, we are optimistic about the recall of the finer granularity. We will seek newer evaluation methods to measure that in future work.

Another internal threat is that the data tested in the experiment discussed in Section 5.3 may not be sufficient. We selected the repositories based on the number of stars on GitHub. Because high-star open-source software (OSS) is used more frequently and has a more demonstrable effect, we believe that the high-star repositories can represent language situations. However, there may be significant differences in project sizes, which brings about the problem of uneven weighting. We will continue to expand the scope of the experiment in the future.

An external threat exists regarding the language extensibility of MSCCD. MSCCD tokenizes code based on parsers generated by ANTLRv4, and there may be parsing errors. If an error occurs at an early stage, for example, during lexical analysis or at a higher position than target subtrees in the PT, MSCCD will not output the expected results. Users must confirm that the used grammar definition file is executable in ANTLRv4 and that it matches the version with the target files.

7 RELATED WORK

Since the 1990s, many code clone detection tools and methods have been proposed. Based on the similarity analysis approach, these tools can mainly be categorized in to four classes: [30] textual [9, 29], lexical [4, 15, 18], syntactic [5, 13, 16], and semantic [10, 19]. The following keywords are commonly found in the research hits:

T3 clone detection: NiCAD [29], iClones [11], Deckard [13], CCAligner [41], and SourcererCC [32] are some well known detection tools that detect code clones up to T3. In the experiment, MSCCD had the same level of recall as these tools. Moreover, Oreo [31] and CloneWorks [37] also have good performance in T3 code clone detection.

T4 clone detection: a T4 code clone is also known as a semantic code clone. AnDarwin [8] uses semantic information to find similar Android applications. SrcClone [2] uses program slicing technology to analyze code segment similarity to detect T4 clones. SCDetector [42] combines the information about the token and the control flow graphs and uses a neural network to generate a code clone detector, achieving very good evaluation results. MSCCD does not have the ability to detect semantic clones and cannot be compared with these tools.

Big-Code clone detection: SourcererCC [32] has increased the scalability of the code clone detection tool to 250 MLOC. Moreover, SAGA [17] detects code clones using a GPU accelerated suffix-array matching algorithm and raises the scalability to the level of 1 BLOC. Due to the amount of calculation, MSCCD is unable to outperform these tools on scalability.

In the era of **multilingual code clone detection**, CCFinderSW [34] had the highest language extensibility before MSCCD was proposed. CCFinderSW generates a source-code parser by converting the grammatical rules of the target language into regular expressions. The parser can remove comments from the source code and generate a token sequence to match CCFinderX 's [14] clone detector. When the comment syntax of the target language cannot be converted into regular expressions, CCFinderSW cannot support it. Additionally, CCFinderSW only supports the detection of T2 clones. Both weaknesses have been resolved in MSCCD. Some text-based tools [9] can also support multiple languages, but the similarity information available in such ancient technologies is too scant to detect more clone types. Furthermore, some existing tools claim to be easy to extend to new languages, but it turns out that the language extensibility of these tools is not enough.

Different from traditional detection tools, **Cross-language code clone detection** aims to detect code cloning between different languages. Perez et al. [25] detected clones between Java and Python by learning token-level vector representations and an LSTM-based neural network. CLCDSA [22] can detect cross-language clones without generating an intermediate representation by learning and comparing the similarity of features. LICCA [40] extracts syntactic and semantic similarities based on the high-level representation of code from the SSQSA platform and can detect clones between five languages, including Java and C. MSCCD does not have the capability of cross-language clone detection. However, we expect that source code normalization with the same language extensibility as MSCCD can be used for cross-language clone detection.

There have been studies that target programs in which multiple languages are mixed [3, 20, 39]. In particular, web systems often contain programs in which multiple languages are mixed [21, 27], and analysis tools for such systems are needed. Nakamura et al. also worked on code clone detection and suggested a detection tool for a web system [23]. Extending MSCCD to include such systems by using techniques such as island grammars [20, 39] is also a future challenge. Note that cross-language code clone detection assumes that each software system is written in a single language and should be distinguished from the case where multiple languages are mixed.

Determining how to benchmark code clone detection tools is a perennial problem for code clone researchers. Initially, applications to large-scale OSSs such as well-known operating systems (e.g., FreeBSD, Linux, and NetBSD) were frequently performed, and subsequent studies were compared by applying them to the same large-scale OSS as prior studies [13, 15, 18]. The first large-scale benchmark for a code clone detection tool was created by Bellon et al. [6]. They visually judged whether a code clone was present. This benchmark is composed of C source code. The most prominent and recently used benchmark is BigCloneBench [35, 36], which was also used in this study. When creating this benchmark, Svajlenko and Roy succeeded in creating a larger benchmark than that of Bellon et al. by performing code mutations. This benchmark comprises Java source code. In 2021, Svajlenko and Roy published a mutation and injection framework for benchmarking using mutation analysis [38]. There is also a growing body of research that uses competitive programming code, such as Google Code Jam, as benchmarks [25, 43, 45]. Much of the research leveraging these benchmarks are deep-learning-based code clone detectors [25, 43, 45]. The Project CodeNet[13], which IBM recently released, will also be used as a benchmark in this research. Existing benchmarks are generally composed of Java or C source code [6, 35]. With the increase in multilingual code clone detection, such as MSCCD, it is expected that benchmarks composed of source code in various languages will be created in the future.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed MSCCD, a grammar-pluggable code clone detection tool. MSCCD showed the highest language scalability and good performance in terms of evaluation metrics, including recall and precision. In a case study with multiple languages, we discovered the impact of language features on code clone detection, revealing that further research in other languages is needed.

In future works, we plan to evaluate MSCCD 's precision in various languages. We also intend to enable MSCCD to cover more clone types while maintaining the same language extensibility and further improving the scalability of MSCCD.

ACKNOWLEDGMENTS

This work was supported by JST, PRESTO Grant Number JPMJPR21PA, Japan. Also, this work was supported by JSPS KAKENHI Grant Numbers JP18H04094 and JP19K20240.

[13] https://github.com/IBM/Project_CodeNet

REFERENCES

[1] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Systematic Review on Code Clone Detection. *IEEE Access* 7 (2019), 86121–86144. https://doi.org/10.1109/access.2019.2918202

[2] Hakam W Alomari and Matthew Stephan. 2020. Srcclone: Detecting code clones via decompositional slicing. In *Proceedings of the 28th International Conference on Program Comprehension*. 274–284.

[3] Alberto Bacchelli, Andrea Mocci, Anthony Cleve, and Michele Lanza. 2017. Mining structured data in natural language artifacts with island parsing. *Science of Computer Programming* 150 (2017), 31–55.

[4] Hamid Abdul Basit and Stan Jarzabek. 2007. Efficient token based clone detection with flexible tokenization. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 513–516.

[5] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance*. IEEE, 368–377.

[6] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591.

[7] Eunjong Choi, Norihiro Yoshida, Raula Gaikovina Kula, and Katsuro Inoue. 2015. What do practitioners ask about code clone? a preliminary investigation of stack overflow. In *Proceedings of the 9th International Workshop on Software Clones*. 49–50.

[8] Jonathan Crussell, Clint Gibler, and Hao Chen. 2015. AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. *IEEE Transactions on Mobile Computing* 14, 10 (2015), 2007–2019.

[9] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings of International Conference on Software Maintenance*. 109–118.

[10] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. 321–330.

[11] Nils Göde and Rainer Koschke. 2009. Incremental clone detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineerin*. 219–228.

[12] ASE Group et al. 2006. Ijadataset 2.0.

[13] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*. 96–105.

[14] Toshihiro Kamiya. 2021. CCFinderX: An Interactive Code Clone Analysis Environment. In *Code Clone Analysis*. Springer, 31–44.

[15] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[16] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. 1996. Pattern matching for clone and concept detection. *Automated Software Engineering* 3, 1 (1996), 77–108.

[17] Guanhua Li, Yijian Wu, Chanchal K Roy, Jun Sun, Xin Peng, Nanjie Zhan, Bin Hu, and Jingyi Ma. 2020. SAGA: efficient and large-scale detection of near-miss clones with GPU acceleration. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering*. 272–283.

[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (2006), 176–192. https://doi.org/10.1109/TSE.2006.28

[19] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 872–881.

[20] L. Moonen. 2001. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering*. 13–22.

[21] Tariq Muhammad, Minhaz F. Zibran, Yosuke Yamamoto, and Chanchal K. Roy. 2013. Near-miss clone patterns in web applications: An empirical study with industrial systems. In *Proceedings of the 26th IEEE Canadian Conference on Electrical and Computer Engineering*. 1–6.

[22] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. In *Proceedings of the 34th International Conference on Automated Software Engineering*. 1026–1037.

[23] Yuta Nakamura, Eunjong Choi, Norihiro Yoshida, Shusuke Haruna, and Katsuro Inoue. 2016. Towards Detection and Analysis of Interlanguage Clones for Multilingual Web Applications. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Vol. 3. 17–18.

[24] Tung Thanh Nguyen, Hoan Anh Nguyen, Jafar M. Al-Kofahi, Nam H. Pham, and Tien N. Nguyen. 2009. Scalable and incremental clone detection for evolving software. In *Proceedings of International Conference on Software Maintenance*. 491–494.

[25] Daniel Perez and Shigeru Chiba. 2019. Cross-language clone detection by learning over abstract syntax trees. In *Proceedings of the 16th International Conference on Mining Software Repositories*. 518–528.

[26] Tom Preston-Werner. [n. d.]. Semantic versioning 2.0.0. https://semver.org/spec/v2.0.0.html

[27] Damith C. Rajapakse and Stan Jarzabek. 2007. Using Server Pages to Unify Clones in Web Applications: A Trade-Off Analysis. In *Proceedings of the 29th International Conference on Software Engineering*. 116–126.

[28] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[29] Chanchal K. Roy and James R. Cordy. 2008. NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th International Conference on Program Comprehension*. 172–181. https://doi.org/10.1109/ICPC.2008.41

[30] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

[31] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 354–365.

[32] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering*. 1157–1168. https://doi.org/10.1145/2884781.2884877

[33] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. 2017. CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference*. 654–659.

[34] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. 2018. Multilingual Detection of Code Clones Using ANTLR Grammar Definitions. *2018 25th Asia-Pacific Software Engineering Conference (APSEC)* (2018), 673–677. https://doi.org/10.1109/apsec.2018.00088

[35] Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating Clone Detection Tools with BigCloneBench. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*. 131–140. https://doi.org/10.1109/icsm.2015.7332459

[36] Jeffrey Svajlenko and Chanchal K. Roy. 2016. BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 596–600.

[37] Jeffrey Svajlenko and Chanchal Kumar Roy. 2017. Fast and flexible large-scale clone detection with CloneWorks.. In *Proceedings of the 39th International Conference on Software Engineering Companion*. 27–30.

[38] Jeffrey Svajlenko and Chanchal K. Roy. 2021. The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis. *IEEE Transactions on Software Engineering* 47, 5 (2021), 1060–1087.

[39] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. 2003. Robust Multilingual Parsing Using Island Grammars. In *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research*. 266–278.

[40] Tijana Vislavski, Gordana Rakić, Nicolás Cardozo, and Zoran Budimac. 2018. LICCA: A tool for cross-language clone detection. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 512–516. https://doi.org/10.1109/SANER.2018.8330250

[41] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*. 1066–1077.

[42] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: software functional clone detection based on semantic tokens analysis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 821–833.

[43] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the 41st International Conference on Software Engineering*. 783–794.

[44] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. 2010. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics* 1, 1-4 (2010), 43–52.

[45] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 141–151.