The Impact of Syntax Trees and Semantic Issues on the Source Code of the Plagiarism Detection Tools

Darko Puflovic[1], Milena Frtunic Gligorijevic[2], Leonid Stoimenov[3], Darko Puflovic
University of Niš
Aleksandra Medvedeva 14, Niš, 18000
Serbia
{darko.puflovic@elfak.ni.ac.rs, milena.frtunic.gligorijevic@elfak.ni.ac.rs, leonid.stoimenov@elfak.ni.ac.rs}

ABSTRACT: *Plagiarism detection is a major exercise in the academic environment for which several tools are deployed. They rely on proper source code and some time, the source code is modified to hide copying. In these cases, the results are distorting. We in this work have used syntactical and semantic aspects of the code to do pre-processing activities. Next, we process the abstract syntax trees to find the correct volume of the location of the plagiarized code.*

Keywords: Plagiarism Detection, Source Code, Abstract Syntax Trees, Intermediate Language, N-gram

DOI: 10.6025/jisr/2020/11/4/99-104

Received: 28 April 2020, Revised 27 July 2020, Accepted 12 August 2020

1. Introduction

With increasing quantity of source code and its availability on the internet there is an increasing need for a system that can detect plagiarism inside newly written ones. A number of solutions that can be used to deal with this problem already exist [1]. These systems use various technologies to identify plagiarism inside specific languages or across variety of different ones.

However, these solutions are somewhat outdated and can't cope with a growing number of features in new and updated versions of programming languages. The majority of compilers these days are open source and provides the tools for syntax and semantic analysis through APIs, so that information obtained through them can provide more detailed and accurate analysis that can assist in the detection of the more complex plagiarism cases.

Syntax and semantic analysis are used to transform syntax trees into more generic representations that are afterwards going to be used to determine similarity between them in combination with the intermediate language code that is obtained by compiling original source. These procedures will be presented on the example of the C# programming language, using .NET

Compiler Platform1 (.NET Compiler Platform - https://github.com/dotnet/roslyn/) , but it can also be used on codes written in other programming languages, like Visual Basic.NET, F#, IronPython, IronRuby and other .NET languages. Also, with minor changes, it can be used for other languages that offer a similar set of tools for syntax and (1 .NET Compiler Platform - https://github.com/dotnet/roslyn/_ semantic analysis, such as Swift, Kotlin and others.

In the next section, an overview of some solutions that are commonly used to detect plagiarism will be given. Section 3 provides information on the benefits of the proposed approach.

2. Related Work

The problem of plagiarism detection occurs in different areas and forms. Using the code without crediting author is a copyright infringement as well as the use of any other material without author's permission. Source code plagiarism may appear in academic environments, but also in companies. Different environments and levels of knowledge of programming languages can bring a diversity of methods that can be used to accomplish same task. This makes the problem of detecting different types of plagiarism difficult.

There are a number of solutions that can cope with various types of plagiarism in different ways [2]. Most of the problems that occur during the task of detection are mainly related to the definition of the plagiarism in source code. The different levels at which they can occur carry different criteria by which it is necessary to determine what can be considered plagia-rism.

Less knowledge of the language syntax diversity offers fewer ways in which that code can be altered and those alterations often represent smaller challenge to the detection system. With increasing knowledge of the possibilities that language offers, there are growing opportunities to hide intention and to make detection process more difficult. Nevertheless, this is not the only problem, because every task set before the developer has a limited number of solutions. It is hard to determine how much change in code is enough to consider code original.

Most of the plagiarism detection tools are designed to work inside academic environment, mostly due to their primary use in reviewing students' work.

Existing plagiarism detection tools [1] use variety of techniques to accomplish that task. Some solutions, like Plaggie [3] can detect similarities across files in one programming language, in this case Java. Few other solutions, namely CodeMatch [4], JPlag [5] and MOSS [6] use different approaches to make sense of plagiarism across large number of languages. Sherlock [7] uses an approach similar to natural language processing. Taking into account all the elements of the syntax,

Sherlock can detect, with a great precision, verbatim copies of code. On the other hand, any change in identifier names, comments or order in which operands appear can be misleading for the system

CodeMatch combines algorithms to match different types of syntax. Numerous programming languages that are supported by this tool give more options to its users, but also give worse results in few cases when knowledge about specific language is of great importance, like when identifier names are changed.

JPlag is well-known online plagiarism detection tool that takes language structure into consideration. The only modifica-tions that represented the challenge for this tool were due to the change in order of the code parts. Those modifications occur mostly during method extractions.

MOSS is another well-known tool with online access. Detection of the places where plagiarism occurs is one possibility. Large number of programming languages are supported, and that is the reason behind bad results in case of complex transfor-mations of the code that can mislead this tool. Like JPlag, more information about specific programming languages are needed to make better judgment.

Most of these solutions are designed to work with a variety of programming languages and examine similar aspects that appear in all of them. However, this approach doesn't give the best results in cases when it is necessary to recognize copied parts of code that are deliberately altered to conceal the intention to use someone else’s code with low possibility of being detected.

There are other attempts to utilize different methods from natural language processing [8], even machine learning [9], but all of these methods are lacking deeper understanding of code semantics. On the other hand, having all the information about syntax and semantics in the code, gives all the information needed for analysis of one programming language, but makes that approach hardly usable on source code written in different language.

3. Proposed Approach

Although the results that we have obtained by using tools described in chapter 2 are good, it is easy to intentionally change source code in order to confuse the tool into thinking that code isn’t plagiarized. Those kinds of changes are easy for human to apply, but hard to spot by an automatic tool. To enable detection of this kind of plagiarism, it is necessary to transform code first into more generic form, and then use comparison methods that have more information about code itself.

.NET Compiler Platform provides access to internal mechanisms of the C# and Visual Basic.NET compilers. In this way, it is possible to access information that compiler use in the translation process of the code into intermediate language and to obtain intermediate language itself for further analysis. Through syntax analysis of the code, it is possible to get information about syntax nodes inside syntax tree, but also the tokens and trivia from the parts of that tree. In most cases, this information is valuable enough for simpler transformations, but more complicated ones require knowledge of semantic code features, like data types, namespaces that contain certain class or list of unused using directives. This information can be provided through semantic APIs, that are part of the compiler. The combination of those two APIs can provide sufficient information necessary to transform original code into the representation that is more suitable for comparison.

Before the analysis of the code, some pre-processing steps have to take place first. The task of pre-processing is to ensure that the codes have the same representation of similar syntax elements so that further analysis process is not disturbed by these changes.

3.1. Pre-Processing of the Source Code

Pre-processing steps, that are going to be described below, include loop and if to switch statement transformations, replacement of unary operators with their expended form, and using of full namespace paths for classes and class members. These transformations were chosen because they are commonly used to conceal plagiarized parts of the code.

There are several implementations of loops in C# programming language, but they all have similar role in the source code. The author may deliberately exchange one type of loop for another to mask the copied part of the code. Using syntactic analysis, it is possible to translate all different kinds of loops in while loop. One possible problem, during this transformation is the scope of the variables defined before the loop. This problem can be solved by placing the loop code inside the block syntax. One more problem is transformation of the foreach loop that does not access elements in the same way as the other loops. It is possible, however, to place the current pointer to the element of the collection inside variable and to call MoveNext method in every iteration of the loop.

Another potential problem for detection of the similarity may be the use of switch statements instead of if, else if or else statements. Transformation of switch statement can be problematic due to the possibility of using jump instructions, but it is possible to eliminate these problems and produce the code that works in the same way as before the change.

Using unary operators it is possible to change syntax tree without modifying program output by adding just a few characters to the code. Just one example of these transformations is the use of plus (+) operator before numeric literal. In most cases, source code contains only minus operator when it is necessary to store negative number, but adding plus sign before number doesn’t change the result, but does change syntax of the code. Another transformation that is necessary to do is to change prefix and postfix increment and decrement operators (i++, i—, ++i, —i) in the full form using plus and minus operators, number literal 1 and equals expression (i = i + 1). This transformation does not have to be the result of a deliberate attempt to mask plagiarism, but also different style that author uses to write his code. One more transformation is the replacement of the assignment with operation into full form (i += 10 into i = i + 10). Operations that can be used inside this expression are plus (+), minus (-), divide (/), multiply (*), modulo (%), and (&), or (|), exclusive or (^), left shift (<>).

Due to the large number of classes, which sometimes have the same name, the use of different namespaces can alter the program behavior. Using directives allow the usage of shorter namespace names that can appear in the code as substitute for longer ones inside these directives. Newer versions of C# pogramming language offer the ability to use “using static” directive that allows all static members of the class to be used directly from code, without specifying class name. All these features make programming easier, but also open the door to possibilities for masking copied parts of the code. The transformation that is part of the pre-processing of the code allow replacement of all methods, delegates, properties, fields, events, classes, structures and interfaces into fully qualified names that include alias (global or extern alias) and whole namespace. In the case of using static directive, class members can be replaced with fully qualified name with the class name they belong. After all identifiers are replaced with fully qualified names, it is possible to remove unnecessary using directives from code. This step is optional since sometimes unnecessary using directives can be valuable for determining if the entire file is a copy.

The list of pre-processing transformations doesn’t end here and contains some steps that mostly deal with literal changes, like replacements of numeric literal values with expression that evaluates the same result (e.g. 40 with 10 + 30) or replacing characters inside string literals with the same Unicode values (e.g. “s” with “\u0073”).

3.2. Similarity Measures

Once the pre-processing transformations are completed, the task of identifying similarities inside the code can begin. This approach uses 3 different techniques to detect plagiarism:

• Comparison of the abstract syntax trees [10, 11]

• Comparison of the source code text

• Comparison of the intermediate language

Transformed syntax trees contain information about whole expressions, nodes and trivia. Syntax nodes and trivia consist of too specific information to be used in detection, but they are part of expressions. The task of comparing the similarities has to start from the comparison of the expressions on the same level and depth of syntax tree.

By comparing the expressing types and specific information about that type of expression it is possible to dismiss many of those that are not similar in any way. Ones that are potentially similar are used in further analysis by comparing syntax nodes and trivia that they consist of. Identifier names may be taken into consideration or not during the comparison, which can help to identify intentional changes to cover up copied code. Another problem that has to be addressed is the usage of extract method refactoring that enables expressions to be moved to another method that can be called from the first one. Using syntax information it is easy to find the location of the original expressions and replace values of parameters inside them with original values from the starting method. In this way, it is possible to identify even those cases with great accuracy. Finding a set of subtrees that contains the same information in two syntax trees can be used to display similarity results in different ways. Two of the simplest ones are using the similarity measure in percentage and by labeling the text representation so the user can see and compare them.

Pre-processed syntax trees can be represented as source code text as well. Using tokenization, that text can be divided that enables the creation of the n-gram models. These models can be further used to calculate probabilities of element occurrence inside the source code. Comparing these probabilities it is possible to conclude what percentage of the code is plagiarized and to show that parts of the code as well. Resolving fully quantified names of identifiers plays crucial role in this step by enabling detection of similar parts of code without a lot of mistakes during the process. Likewise, this reduces the impact of the non-similar parts, because it dramatically increases the number of tokens in the code. Anyway, this comparison provides insight into similarities which are located in different parts of the code which can be useful in cases of extracted methods, when the code is not in the same method in original and plagiarized versions of the code.

A previous method work great on the text representation of the code, but don’t really help identifying similarities in results and are related to the programming language C#. Comparison of the intermediate language allows these two issues to be addressed. Intermediate code (IL) generated from any .NET language can be represented using the similar set of IL instructions. Those instructions are emitted after compiler optimizations that make them even more suitable for analyzing output results of the code. IL instruction consists of the label, instruction and, if there is a need, argument (e.g. IL_0000: ldc.i4.3). Creating n-gram model out of the pair that consists of instruction and argument can be used to detect similar parts of the code and show that similarity in the same way as the previous approaches.

3.3. Results

CSPlag, described in this paper was tested on four pairs of source codes written for this purpose. Results of these comparisons are shown in Table 1 and will be discussed in the following paragraphs.

Tool	**1**	**2**	**3**	**4**
Expected results	100%	50%	15%	0%
JPlag	100%	25.8%	26%	NA
MOSS	53%	NA	NA	NA
CSPlag	100%	43%	11%	4%

Table 1. Comparison With Jplag And Moss

The first pair of codes contains identical expressions with altered identifier names, one extracted method and using few labels and jumps. JPlag proved to be very reliable in this task, but MOSS had troubles, stopping at the first jump without detecting rest of the copied code.

The second pair consists of two different source codes with copied loops that are exchanged in second file into different kind of loop (for loop into while and foreach) and if, else statements are replaced by switch statement in the secondfile. JPlag recognized some statements but had issues recognizing loop replacements and similarities between if, else statements and their switch substitute. MOSS, on the other hand, didn’t recognize similarity sufficiently large to notify the user about potential plagiarism. Pre-processing techniques that CSPlag use had impact on result which is higher than in the first two systems.

The third pair of codes contains few statements that have the same name but represent calls to different methods and few others that use different name for namespace identifier, but represent the same namespace. In this case JPlag recognizes all method calls, regardless of what namespace they belong, as the same method call. The result reflects the situation by showing slightly higher percent of similarity then there is in the code. MOSS on the other hand gives no results in this case again, because of the altered namespaces in the calls. CSPlag detected these changes and didn’t recognize these pieces of code as similar ones which caused lower similarity.

Finally, last pair consists of different codes and results in all three approaches gave good results on this task.

4. Discussion and Conclusion

Applying pre-processing to the source code and syntax trees proved to be very successful and of great importance in the overall similarity detection. Some refactoring tools that come preinstalled in development environments can obstruct the task of plagiarism detection. Those tools use syntax and semantic analysis to change original code, so the easiest way to counteract their effect is to use the same techniques to transform code back into original state.

In cases where the developer intentionally changes the parts of the code to cover traces of copied parts, those transformations can provide considerably better similarity results. Similar syntax and semantic information is of great help even in the process of syntax tree comparison, where it is possible to exclude irrelevant information for even better accuracy. In combination with intermediate language comparison, precision of the plagiarism detection can be even more improved and in addition to that, it allows comparison of the codes written in different languages.

On the other hand, transformations are not mandatory and similarity can be obtained even without doing any pre-processing steps. List of pre-processing steps is not limited to ones described in this paper and it is possible to write new ones that can deal with some other forms of plagiarism that are not the part of the system today.

References

[1] Martins, V. T., Daniela, F., Rangel, H. P., Daniela, C. (2014). Plagiarism Detection, A Tool Survey and Comparison, 3rd Symposium on Languages, *Applications and Technologies*, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, vol. 38, pp. 143-158, 2014.

[2] Agrawal, M., Sharma, D.K. (2016). A State of Art on Source Code Plagiarism Detection. *In*: 2nd International Conference on Next Generation Computing Technologies, p. 236-241, 2016, doi: 10.1109/NGCT.2016.7877421.

[3] Ahtiainen, A., Surakka, S., Rahikainen, M. (2006). Plaggie: GNULicensed Source Code Plagiarism Detection Engine for Java Exercises, 6th Baltic Sea conference on Computing Education Research: Koli Calling 2006, p. 141-142, 2006.

[4] Modiba, P., Pieterse, V., Haskins, B. (2016). Evaluating Plagiarism Detection Software for Introductory Programming Assignments, *In*: Computer Science Education Research Conference, p. 37-46, ACM, 2016.

[5] Prechelt, L., Malpohl, G., Phlippsen, M. (2000). JPlag: Finding Plagiarisms Among a Set of Programs, *Technical report*, Fakultat for Informatik, Universitat Karlsruhe, 2000.

[6] Schleimer, S., Wilkerson, D. S., Aiken, A. (2005). Winnowing: Local Algorithms for Document Fingerprinting, 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03), New York, NY, USA, p. 76-85, 2003.

[7] Hage, J., Rademaker, P., Vugt, N. (2010). A Comparison of Plagiarism Detection Tools, Utrecht University, Utrecht, Netherlands, p. 28, 2010.

[8] Chilow. icz, M., Duris, E., Roussel, G. (2009). Syntax TREE Fingerprinting: A Foundation for Source Code Similarity Detection, 2009.

[9] AlSallal, M., Iqbal, R., Amin, S., James, A., Palade, V. (2016). An Integrated Machine Learning Approach for Extrinsic Plagiarism Detection, *In*: 9th International Conference on Developments in eSystems Engineering (DeSE), p. 203-208, Liverpool, United Kingdom, 2016.

[10] Kikuchi, H., Goto, T., Wakatsuki, M., Nishino, T. (2015). A Source Code Plagiarism Detecting Method Using Sequence Alignment with Abstract Syntax Tree Elements, *International Journal of Software Innovation (IJSI)*, p. 41-56, 2015.

[11] Chilowicz, M., Duris, E., Roussel, G. (2009). Syntax Tree Fingerprinting for Source Code Similarity Detection, *IEEE 17th International Conference on Program Comprehension*, Vancouver, Canada, p. 243-247, 2009.