Detecting source code reuse across programming languages

Detección de reutilización de código fuente entre lenguajes de programación

Enrique Flores, Alberto Barrón-Cedeño, Paolo Rosso and Lidia Moreno
Depto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
{eflores, lbarron, prosso, lmoreno}@dsic.upv.es

Resumen: Con el crecimiento de la Web muchas comunidades de programadores ponen a disposición pública códigos fuente bajo licencias que protegen la propiedad de sus creaciones. Es una gran tentación reutilizar un código disponible en la Web que funciona y ya está testeado; en el ámbito académico se observa más esta situación, donde un grupo de alumnos tienen asignada la misma tarea. Un programador puede obtener un código en un lenguaje de programación en el que no está trabajando y traducirlo a otro lenguaje. En este trabajo se proponen dos modelos basados en n-gramas de carácteres para detectar la similitud y posible reutilización de código fuente, incluso tratandose entre códigos escritos en distintos lenguajes de programación. Ambos modelos abordan el problema de reutilización, uno trabaja a nivel de documento, y el segundo trabaja comparando fragmentos de código con el fin de detectar solo partes del código, el segundo representa mejor las situaciones reales de reutilización.
Palabras clave: Reutilización de código, análisis de código fuente entre lenguajes, detección de plagio

Abstract: With the growth of the Web many programmer communities make publicly available source codes under licences that protect their property. For a programmer the temptation of reusing a working source code, available on the Web already tested, could be great. As well this kind of temptation exists in the academic environment where a group of students is assigned the same task. A programmer may obtain a source code in a programming language in which s/he does not work and could translate it into another language. In this work we propose two models based on character n-grams in order to tackle the problem of cross-programming language reuse of source code at document and fragment levels. In the second model, fragments of source codes are compared with the aim of detecting only those fragments in the source code that resemble more real cases of reuse.
Keywords: Source code reuse, cross-language source code reuse analysis, plagiarism detection

1 Introduction

In the digital era, massive amounts of information are available causing the material from other people to be exposed to reuse. Therefore, there is high interest in identifying whether or not a work has been reused. As for documents in natural language, the amount of source code in Internet is huge, facilitating the reuse of all or part of previously implemented programs. Software developers could be tempted to reuse source code. In case of not giving the reference of the original work, plagiarism[1] would be comitted.

As a countermeasure, different models for the automatic detection of source code reuse and a plagiarism have been developed (Cunningham and Mikoyan, 1993; Jankowitz, 1988; Faidhi and Robinson, 1987; Wise, 1992; Rosales et al., 2008).

Cross-language reuse detection has been approached just recently. Let L_1 and L_2 be

[1]Source code reuse is often allowed, thanks to licenses as those of Creative Commons (http://creativecommons.org/)

two programming languages ($L_1 \neq L_2$), we define cross-language source code reuse as the translation of (part of) a source code document d in the language L_1 into the document dq in the language L_2. As for texts written in natural language (Potthast et al., 2011), detecting code reuse when a translation process occurred, is even more challenging; it is very likely that d_q does not represent an exact translation of d because of implementation issues.

As far as we know the only approach that aims to detect cross-language source code reuse is that of (Arwin and Tahaghoghi, 2006). Instead of processing source code, this approach compares intermediate language (RTL) produced by a compiler. The comparison is in fact monolingual and compiler dependent. Unfortunately, the corpus in this research work used is not available, making the direct comparison to this approach unfeasible.

Our contribution represents an attempt to detect source code reuse among three programming languages: C++, Java and Python on the basis of natural language processing techniques.

The remainder of this paper is structured as follows. Section 2 is dedicated to review the different approaches that treat the detection of source code reuse. In Section 3 we describe the source code document model whereas in Section 4 we present the novel sliding window based model. In Section 5 we illustrate the experiments we carried out and we discuss the obtained results. Finally, in the last section we draw some conclusions and discuss future work.

2 State of the art

Two are the kinds of approaches used for detecting source code reuse. The Fist approach is based on the count of certain attributes of source code, as in the pioneering work of Halstead (Halstead, 1972).

Theirs method uses attributes such as the number of operands, number of operators, and the different number of operands, among others. Similar is the work of Selby (Selby, 1989) although different are the attributes used: computacional time and usefulness of a part, number of static calls, etc. A second kind of approaches for the detection of source code reuse, which is the most used up to date, focuses on the implementation structure of the code to determine the existence of similarity. For this reason, most works create a profile of that structure. Jankowitz (Jankowitz, 1988) proposes to create a profile of the execution tree and to perform an in-order visit representing leaves with '1' and ramifications with '0' to allow identifying similarities between codes efficiently. The most important work in this research line is the one of Whale (Whale, 1990b) where branches, repeats and statements, are codified for finally comparing codes according to the difference between these codifications. Several are the tools that have been inspired by the above method, such as Plague (Whale, 1990a) and its further developments YAP1, YAP2 and YAP3 (Wise, 1992).

Another tool that deserves to be mentioned is JPlag (Prechelt, Malpohl, and Philippsen, 2002). JPlag is able to detect source code reuse in different programming languages although at monolingual level, that is, one programming language at a time. It takes into consideration syntax and tokens from source code, looking for common strings between programs. Another interesting work is the MOSS project (Schleimer, Wilkerson, and Aiken, 2003), which uses the technique of fingerprinting and winnowingfor detecting similarities in parts of the code. Cunningham (Cunningham and Mikoyan, 1993) presents Cogger, a tool which uses Case-Based Reasoning to generate a profile of a program in order to find common sequences. Other strategies that are widely used in the academic environment are those based on mutual information between programs, as show in Zhang et al. (Zhang, Zhuang, and Yuan, 2007). This last work attempts to detect several levels of obfuscation[2]; they generally achieve better results than similarity detection tools such us JPlag and MOSS.

The PK2 tool by Rosales et al. (Rosales et al., 2008), focuses the detection of source code reuse on the search of common subsequents of longer strings. It has been tested on four different corpora obtaining quite encouraging results. Unfortunately, in cases of short programs such as assembly language the tool failed. Finally, we highlight the work of Burrows et al. (Burrows, Tahaghoghi, and Zobel, 2006), whose aim is to search for suspicious fragments of source code in large reposito-

[2]Obsfuscation in plagiarism can be considered as noise insertion.

ries. To identify source code reuse in large repositories, they employ techniques such as tokenization of terms and reverse list n-grams (where each n-gram shows the number of documents that contain it), the individual document identifiers, and the number of occurrences of the term in each document.

Most of the studies on source code reuse have been done to detect cases of reuse in the same programming language. Although a preliminary attempt was made by Halstead (Halstead, 1972), in his research work he tries to find a common formula to calculate the similarity of the different implementation of an algorithm in several programming languages. As already mentioned, Arwin and Tahaghoghi (Arwin and Tahaghoghi, 2006) compare source code using the intermediate language generated by a compiler (RTL). The tool, Xplag, allows to detect similarities between source code written in different programming languages. They created a programs collection obtained by translating programs written in the C language into Java language using the Jazillian online tranlation tool to simulate the way students may copy.

Finally, we mention that in many of the previous works authors preprocessed source code making changes such as removing comments, case folding, changing synonyms into a canonical form (e.g. if instructions where a value is evaluated against another and can be represented by $>$, $\geq$, $<$ o $\leq$, are converted into a unique form in order to compare under the same pattern; or *while* instructions are transformed into *for*, etc.), reordering the definition of the functions as shown in (Whale, 1990a) and (Wise, 1992), removing a certain percentage of reserved words of the programming language that are not relevant as, for instance, the PK2 tool (Rosales et al., 2008) does; or converting all the words into symbols (i.e., main $\rightarrow$ B) and after the conversion making n-grams of characters, and working with words as implicitly proposed in Burrows et al. (Burrows, Tahaghoghi, and Zobel, 2006). The experiments are performed under different conditions of preprocessing as deleting comments or keywords, among others, in order to investigate a number of possibilities to effectively solving the problem of the detection of cross-language source code reuse.

The approach described in this paper is different from those previously mentioned because it is not based on the structure defined in the code neither in the occurrence of attributes. It is based on the information the user provides in the code such as variable names, how many times a type of function is used, or comments in natural language, among others.

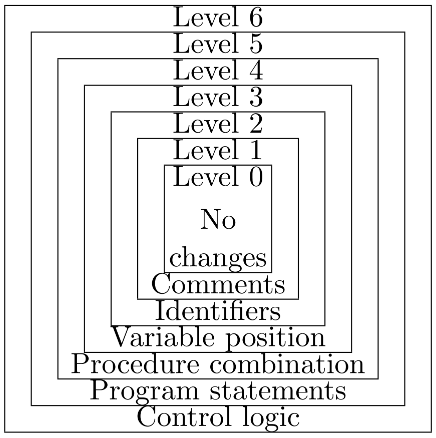

Figure 1: Levels of program modifications in a plagiarism spectrum.

3 Source code document model

Our first attempt to address the types of modifications that the user can make in order to obfuscate the source code reuse, was made accordingly to the levels of the program modifications discussed by Faidhi and Robinson (Faidhi and Robinson, 1987) and ilustrated in Figure 1.

We aim to treat some of these levels of program modifications, considering: (*i*) full code, i.e., source code and comments, for level 0; (*ii*) full code without comments (fc-wc) for level 1; (*iii*) programming language reserved words only (rw only) for levels 2 and 3. Additionally, three more exploratory experiments have been carried out: (*iv*) comments only (*v*) full code without reserved words (fc-wrw) and (*vi*) full code without comments and without reserved words (fc-wc- wrw).

The proposed source code document model is composed the following three modules:

(*a*) *Pre-processing*: line breaks, tabs and spaces removal as well as case folding;

(*b*) *Features extraction*: character n-grams extraction, weighting based on normalised term frequency (tf);

(*c*) *Comparison*: cosine similarity estimation on the basis of the formula below:

$$cos(d, d_q) = \frac{\Sigma_{t \epsilon d \cap d_q}(tf_{t,d} \cdot tf_{t,d_q})}{\sqrt{\Sigma_{t \epsilon d}(tf_{t,d})^2 \cdot \Sigma_{t \epsilon d_q}(tf_{t,d_q})^2}} \; . \quad (1)$$

In Equation 1, the reference document is represented as d, the suspicious document as dq and n-gram terms as t. The result of the equation is in the range [0-1].

Once d_q is compared to $d \in D$ (where D represents the set of source code documents), a sorted list is generated in order to rank the potential sources for the suspicious program d_q. The top k pairs (d_q; d) in the ranked list are the most similar and, therefore, more likely to be reused.

4 Source code sliding window model

One of the problems we could potentially face with the previous model, is that only a part of source code could be reused and, therefore, comparing source code at document level would not allow us to detect it.

In this section we propose a novel model that on the basis of a sliding window is able to compare source code at fragment level as illustrated in Figure 2. Two sliding windows wd and wdq of the same size s are scrolled with distance l along the two source code documents d and dq. The weighting for feature extraction is based on tf and the comparison between wd and wdq is done using cosine similarity.

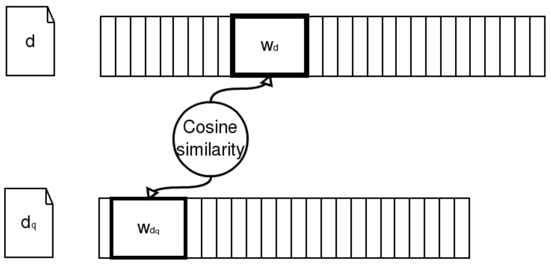

Figure 2: Example of the source code sliding window model.

Each window wd of the reference document d is compared with all the windows w_{d_q} obtained scrolling them along the suspicious document d_q. As a result of all comparisons between the two documents, a matrix of values representing the similarity between the different fragments is obtained. Those values that exceed a threshold t are considered for the similarity estimation between the two source codes.

Therefore, the proposed source code sliding window model is composed as the previous model of three modules:

(*a*) *Pre-processing*: line breaks, tabs and spaces removal as well as case folding;

(*b*) *Features extraction*: on the basis of the sliding windows, character n-grams are extracted using tf ;

(*c*) *Comparison*: fragment cosine similarity estimation.

With respect to the source code document model this model extracts features from the fragments of the source documets. Moreover, the comparison of source code is made at fragment level and not at document level.

5 Experiments and results

Due to the lack of cross-language source code corpora[3], in order to compare the two proposed models we had to develop a corpus composed of programs in different programming languages. This corpus, named SPADE corpus, includes codes in C++, Java and Python[4]. For each language a collection of programs exist that maintains a correspondence to the programs in the other languages. The collections in C++ and Java have been partially reused. The cases Python→C++ represent real examples of cross-language reuse. The cases Python→Java represent simulated cases. Moreover, the cases Java–C++ represent triangular reuse (having Python as pivot). Table 1 shows some statistics of the corpus.

SPADE Corpus. It is a collection of documents that contain programs in C++, Java and Python in a multi-agent system, which allow to develop agents with their own behavior. For each language we have a collection of programs that have some correspondence with programs in other languages This correspondence may be total or partial functionality.

- **Python API**. It is the largest part of the corpus, which incorporates the com-

[3]The authors of the corpus used in (Arwin and Tahaghoghi, 2006) have been contacted but the corpus seems not to be avaliable anymore.

[4]The corpus is available for research porpouses at url: http://users.dsic.upv.es/ eflores/

Language	Tokens	Avg. length of tokens	Types	Types per program	Programs
C++	1,318	3.46	144	28.8	5
Java	1,100	4.52	190	47.5	4
Python	10,503	3.24	671	167.75	4

Table 1: Statistics of the SPADE corpus.

plete system to to develop agents communication. Consists of the total number of documents that contain programs have been used only 4 with 10503 tokens in total, because they are having similar functionality to those written in other languages. This code has been developed by SPADE project members.

- **C++ API**. It consists of 5 documents that contain programs in C++, with a total of 1318 tokens. These programs have the same functionality of the programs written in Python but they have been developed independently. This part was developed by students of IARFID M.Sc..
- **Java API**. This part consists of 4 documents with a total of 1100 tokens. Java programs that contains the result of the translation of the part written in C++ developed with the intention to be as similar as possible.

5.1 Experiments at document level

In order to test the first pourpose source code document model we have considered diferent sizes of n-grams: n={1,. . . ,5}. Table 2 shows the average and standard deviation of the positions of the referent document d with respect the suspicious document d_q for the best results obtained using n=3. In the most of the experiments the best result is obtained when considering *full code* as well as *full code without comments* with the same values in both cases. The best results obtained with n=3 are in line with those for natural language documents described in (Potthast et al., 2011).

From the obtained results, comments in source code seem not to have much impact, partly because programmers could have decided to rewrite them, to write their own comments, or simply not to take into account the previous comments when reusing the source code. Therefore, it makes sense to ignore comments during the comparison of source code reuse. Moreover, a malicious programmer could modify the comments to introduce noise in the detection.

With respect to the comparison across programming languages, the best results are obtained for C++ and Java source codes because of their syntax and vocabulary similarity.

5.2 Experiments at fragment level

In order to test the novel source code sliding window model we have used the following ranges:

- window size: s={10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300}
- distance: l={10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300}
- threshold: t={0.1,. . . ,0.9}

Due to that some source code after preprocessing did not exceed 300 characters, we have not considered using window sizes or distances greater than 300 characters.

Several experiments have been performed: (*i*) with overlap between subsequent sliding windows ($l < s$); (*ii*) without overlap between subsequent sliding windows ($l = s$); (*iii*) with gap between subsequent sliding windows ($l > s$).

Figure 3 shows that such as overlapping between subsequent sliding windows has not helped to improve the results.

In Figure 4 we illustrate the improvement that we have obtained for the Java and C++ example with different threshold values. Similar findings have been found for the others program languages pairs. Therefore, whereas the distance between subsequents sliding windows seem no to be usefull the threshold value helps to obtain better results.

In order to compare the different pair of languages, several combinations parameters need to be taking into account. As previously shown, the distance between subsequent sliding windows does not help to improve performance. Therefore, in order to minimize

Features	Java – C++	Python → C++	Python → Java
full code	1.00 ± 0.00	1.44 ± 0.83	1.62 ± 1.10
fc-without comments	1.00 ± 0.00	1.44 ± 0.83	1.62 ± 1.10
fc-reserved words only	1.56 ± 0.83	1.78 ± 1.02	1.75 ± 0.83
comments only	2.29 ± 1.57	2.83 ± 1.34	3.00 ± 0.67
fc-without rw	1.44 ± 0.83	1.78 ± 1.13	2.00 ± 1.32
fc-wc-wrw	1.44 ± 0.83	1.67 ± 0.94	1.44 ± 0.69

Table 2: Source code document model: results obtained with character 3-grams. Values represent the average and standard deviation of the position of the source code in the ranked list.

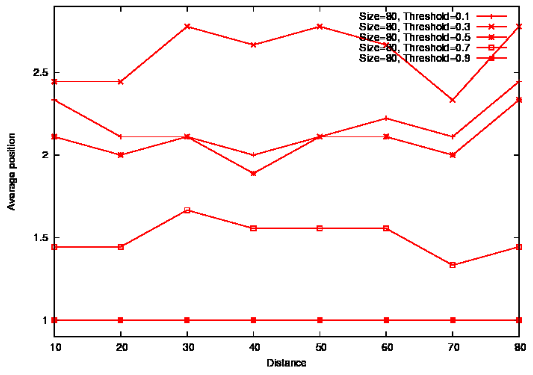

Figure 3: Average position of the source code in the ranked list obtained using fix value for the sliding window size s, varying the threshold t and the distance l between subsequent sliding windows.

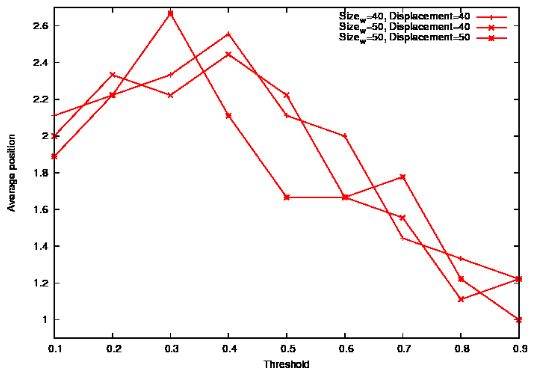

Figure 4: Example of the improvement of the threshold between Java and C++.

the number of comparisons between sliding windows on the two source code documents, we choose the largest distance between subsequent sliding windows among those we obtained the best results with.

Finally, in case of the range of thresholds allows to obtain the best results, we choose the smallest one.

In total we carried out 1.764 experiments taking into account all the combinations of the following parameters: 14 values of distance between subsequent sliding windows (l) * 9 thresholds (t) * 14 sliding window sizes (s). In Table 3 we illustrate the parameter values for l, t and s that allowed to obtain the best result across the programming languages studied. As sliding window size for each programming language pair we have choosen the smallest one that allowed to obtain the best results.

From the table we can observe that the distance between subsequent sliding windows is equal to their size. This means that the best results have been obtained without any overlapping between subsequent sliding windows[5].

With respect to threshold parameter, it is important to notice how hight needs to be in order to allow the detection of source code reuse in cases like Java–C++ programming language pairs. This is due to they share a similar syntax, and, therefore lowest values would not allow to detect any cross-language source code reuse.

Comparing the results in terms of document versus fragment (sliding window) source code models, the average position of the source code in the raked list has been improved for the Python Java pair (1.375 vs. 1.444) and maintains for the other programming language pairs.

6 *Conclusions and future work*

This work is a preliminary attempt to detect cross-language source code reuse. The proposed source code models are based on

[5]Similar results were obtained with the same sliding window size and threshold although with smaller distance between subsequent sliding windows. On the basis of what described in this section, the larger distance value was chosen.

Pairs of languages	Size (s)	Distance (l)	Threshold (t)	Avg. position
Java – C++	50	50	0.8	1.000
Python → C++	90	50	0.1	1.375
Python → Java	30	30	0.2	1.444

Table 3: Best results obtained using the source code sliding window model.

similarity computations at character n-grams level. When similarity computation is at document level (source code document model), the impact of comments, variable names, and reserved words of the different programming languages has been investigated. The best results are obtained when comments are ignored. This suggests that the comments can be safely discarded when aiming to determine the cross-language similarity between two programs. Presumably, the character 3-grams are able to represent programming style as in the case of documents written in natural language. The fact that suggests the best results have been obtained without considering comments and employing trigrams suggest that the simple approach based on character n-grams allows for detecting programming style as like in the case of documents written in natural language as discussed in (Stamatatos, 2009).

When the similarity computation has been calculated at the fragment level (source code sliding window model), we have detected that programming pairs that have a similar syntax, we need to use a higher threshold than programming languages pairs that have not similar syntax. Also we have observed that when we have obtained the same value with differents distances, we have chosen the largest distance between subsequent sliding windows because that reduces the computational cost.

As future work, we identify the following research lines: (i) after adjusting the parameters of the size of the sliding windows, the distance between of the subsequent sliding windows and a the threshold (s, l and t), the next step will be to locate the reused fragments in order to indicate the exact parts where the source code has been reuse; (ii) applying cross-language alignment-based similarity analysis (Pinto et al., 2009), that recently (Potthast et al., 2011) allowed to obtain good results on natural language text, to source codes of different programming language pairs.

7 *Acknowledgments*

This research work was done in the framework of Marie Curie actions PEOPLE-IRSES 269180 WiQ-Ei: Web information Quality Evaluation initiative, MICINN project TEXT-ENTERPRISE 2.0 TIN2009-13391-C04-03 (Plan I+D+i) and VLC/CAMPUS Microcluster on Multimodal Interaction in Intelligent Systems.

References

Arwin, C. and S.M.M. Tahaghoghi. 2006. Plagiarism detection across programming languages. *Proceedings of the 29th Australian Computer Science Conference*, 48:277–286.

Burrows, S., S.M.M. Tahaghoghi, and J. Zobel. 2006. Efficient plagiarism detection for large code repositories. *Software Practice and Experience*, 37:151–175, September.

Cunningham, P. and A. Mikoyan. 1993. Using cbr techniques to detect plagiarism in computing assignments. *Department of Computer Science, Trinity College, Dublin (Internal Report)*, September.

Faidhi, J. and S. Robinson. 1987. An empirical approach for detecting program similarity and plagiarism within a university programming enviroment. *Computers and Education*, 11:11–19.

Flores, E., A. Barrón-Cedeño, P. Rosso, and L. Moreno. In Press. Towards the detection of cross-language source code reuse. *NLDB*.

Halstead, M. H. 1972. Naturals laws controlling algorithm structure? *SIGPLAN Noticies*, 7(2), February.

Jankowitz, H. T. 1988. Detecting plagiarism in student pascal programs. *The Computer Journal*, 31(1).

Pinto, D., J. Civera, A. Barrón-Cedeño, A. Juan, and P. Rosso. 2009. A statistical approach to crosslingual natural language tasks. *Journal of Algorithms*, 64(1):51–60.

Potthast, M., A. Barrón-Cedeño, B. Stein, and P. Rosso. 2011. Cross-language plagiarism detection. *Language Resources and Evaluation, Special Issue on Plagiarism and Authorship Analysis*, 45(1):45–62.

Prechelt, L., G. Malpohl, and M. Philippsen. 2002. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016–1038.

Rosales, F., A. García, S. Rodríguez, J. L. Pedraza, R. Méndez, and M. M. Nieto. 2008. Detection of plagiarism in programming assignments. *IEEE Transactions on Education*, 51(2):174–183.

Schleimer, S., D. S. Wilkerson, and A. Aiken. 2003. Winnowing: Local algorithms for document fingerprinting. *ACMSIGMOD Conference*, pages 76–85, June.

Selby, R. W. 1989. Quantitative studies of software reuse. *Software Reusability*, 2:213–233.

Stamatatos, E. 2009. Intrinsic plagiarism detection using character n-gram profiles. *Proceedings of SEPLN'09, Donostia, Spain*, pages 38–46.

Whale, G. 1990a. Identification of program similarity in large populations. *The Computer Journal*, 33(2).

Whale, G. 1990b. Software metrics and plagiarism detection. *Journal of Systems and Software*, 13:131–138.

Wise, M. J. 1992. Detection of similarities in student programs: Yaping may be preferable to plagueing. *Proceedings of the 23th SIGCSE Technical Symposium.*

Zhang, L., Y. Zhuang, and Z. Yuan. 2007. A program plagiarism detection model based on information distance and clustering. *Internacional Conference on Intelligent Pervasive Computing*, pages 431–436.