DeSoCoRe: Detecting Source Code Re-Use across Programming Languages*

Enrique Flores, Alberto Barrón-Cedeño, Paolo Rosso, and Lidia Moreno
ELiRF, Departament of Information Systems and Computation
Universidad Politécnica de Valencia, Spain
{eflores,lbarron,prosso,lmoreno}@dsic.upv.es

Abstract

Source code re-use has become an important problem in academia. The amount of code available makes necessary to develop systems supporting education that could address the problem of detection of source code re-use. We present the DeSoCoRe tool based on techniques of Natural Language Processing (NLP) applied to detect source code re-use. DeSoCoRe compares two source codes at the level of methods or functions even when written in different programming languages. The system provides an understandable output to the human reviewer in order to help a teacher to decide whether a source code is re-used.

1 Introduction

Identifying whether a work has been re-used has received increasing interest in recent years. As for documents in natural language, the amount of source code on Internet is increasing; facilitating the re-use of all or part of previously implemented programs.[1] If no reference to the original work is included, plagiarism would be committed. The interest for detecting software re-use is great discouraging academic cheating.

Many online tools exist for detecting re-use in text, such as Churnalism[2]. To the best of our knowledge the unique online service to detecting re-use in source code is JPlag[3]. This tool can process different programming languages, but at monolingual level.

This paper presents the DeSoCoRe tool for detection source code re-use across programming languages. We estimate the similarity between two source codes independently of the programming language using NLP techniques. In fact, programming languages are similar to natural languages; both can be represented as strings of symbols (characters, words, phrases, etc.).

DeSoCoRe aims at supporting a reviewer in the process of detecting highly similar source code functions. It allows to visualize the matches detected between two source codes d and d_q. The programs are represented as a graph. An edge exists between a function in d_q and a function in d if re-use between them is suspected. The code chunks are displayed to the user for further review. With the information provided, the reviewer can decide whether a fragment is re-used or not.

2 Related Work

In the previous section we mention only one online tool but many research works for source code re-use detection exist. Two main approaches have been explored: content-based and structure-based.

Content-based approaches are based on analysis of strings within the source codes. The pioneering work of (Halstead, 1972) is based on units countings. He counts the total number of operands, total number of different operands and number of operators, among others.

* Screencast available at: *http://vimeo.com/33148670*. The tool is available at: *http://memex2.dsic.upv.es:8080/DeSoCoRe/*

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

[2] *http://churnalism.com/*

[3] *https://www.ipd.uni-karlsruhe.de/jplag/*

Proceedings of the NAACL-HLT 2012: Demonstration Session, pages 1–4,
Montréal, Canada, June 3-8, 2012.

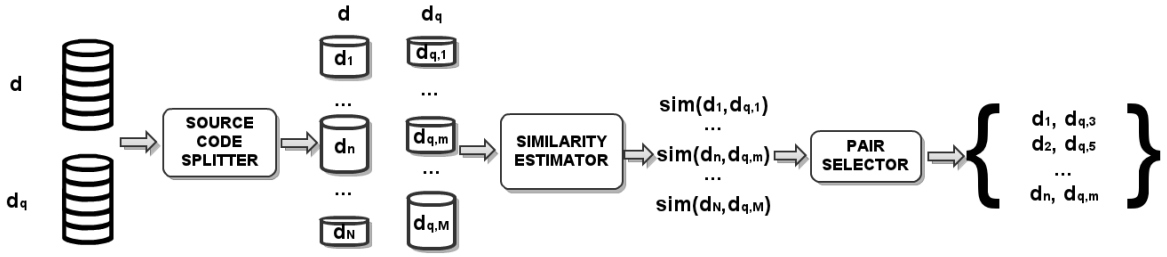


Figure 1: Architecture of DeSoCoRe tool. The source code d has N functions, and d_q has M functions. Each function of d is compared against all the functions of d_q.

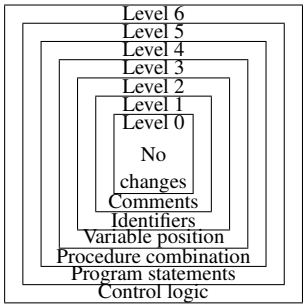


Figure 2: Levels of program modifications in a plagiarism spectrum proposed by Faidhi and Robinson.

Structure-based approaches, the most used up to date, focus the analysis on the code structure (execution tree) in order to estimate the level of similarity between two source codes. A seminal model is the proposed by (Whale, 1990b). This approach codifies branches, repeats, and statements in order to estimate the similarity between two programs. This model has inspired several other tools, such as Plague (Whale, 1990a) and its further developments YAP[1,2,3] (Wise, 1992).

JPlag (Prechelt et al., 2002) combines both approaches. In the first stage, it exploits syntax in order to normalize variables and function names. In the second stage, it looks for common strings between programs. This work attempts to detect several levels of obfuscation[4]. It achieves better results than JPlag for highly obfuscated cases but worst results with low degree of obfuscation.

JPlag is able to detect source code re-use in different programming languages although at monolingual level; that is, one programming language at a time. None of the reviewed approaches is able to perform cross-language analysis. To the best of our knowledge the only approach to analyze cross-language source code re-use is the one of (Arwin and Tahaghoghi, 2006). Instead of processing source code, this approach compares intermediate code produced by a compiler which includes noise in the detection process. The comparison is in fact monolingual and compiler dependent. The resulting tool, Xplag, allows to compute similarity between codes in Java and C.

3 Architecture

As shown in Figure 1, DeSoCoRe consists of three general modules. As input user gives a pair of source codes (d, d_q). The source code splitter is responsible for dividing the codes in functions. To split each code into functions we have developed syntactic analyzers for Python and for C syntax family language: C, C++, Java, C#, etc.

The next module compares the functions of d_q against the functions of d. To make this comparison we have divided the module into three sub-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 normalized term frequency (tf); and (c) *Comparison*: a cosine similarity estimation. As output, we obtain a similarity value in the range [0-1] for all the pairs of functions between the source codes.

We carried out several experiments in order to find the best way to detect re-use in source codes. These experiments were inspired by what proposed in (Faidhi and Robinson, 1987). They describes the modifications that a programmer makes to hide the re-use of source code as Figure 2 shows. These levels are: (i) changes in comments and indentation;

[4]Obfuscation in re-use can be considered as reformulation, which inserts noise.

(*ii*) changes in identifiers; (*iii*) changes in declarations; (*iv*) changes in program modules; (*v*) changes in the program statements; (*vi*) changes in the decision logic. As result of these experiments we obtained best configuration of our system to use the entire source code and to apply 3-grams (Flores et al., 2011).

Once the similarity value has been calculated for all the possible pairs, the pair selector decides what pairs are good source re-used candidates. This module has to discard the pairs which have obtained a similarity value lower than a threshold established by the user. As output DeSoCoRe returns the suspicious pairs that have been re-used.

4 Demonstration

In order to interact with our developed system, we provide a Java applet interface. It is divided in two interfaces: (*i*) input screen: which allows the user for inserting two source codes, select their programming language and additionally to select a value for the similarity threshold;[5] (*ii*) output screen: which shows the results divided in two sections: (*a*) a graphical visualization of the codes; and (*b*) a plain text representation of the codes. In the first section we have used the Prefuse Library[6] in order to draw a graph representing the similarity between the functions of the source codes. The painted graph consists of two red nodes which represent each source code. Their functions are represented by purple nodes and connected to the source code node with edges. If any of these functions has been selected by the system as re-used, its nodes will be connected to a node from the other source code.

Finally, a node is marked in red if it composes a potential case of reuse. When a function is pointed, the plain text section displays the source code. Also, if this function has any potential case of re-use, the function and the potential re-used function will be shown to perform a manual review of the codes. In order to be introduced to DeSoCoRe an example is provided and can be accessed clicking on the *Example* button. Figure 3 shows an example of two supicious source codes: one in C++ and one in Java. The user is able to start the estimation of similarity clicking on *Estimate!* button.

After similarity estimation, the result is displayed as in Figure 3(a). For exploratory purpouses, example source codes are available through the *Example* button. The user is able to start the estimation of similarity clicking on *Estimate!* button. Figure 3(b) shows an example of potential cases of reuse. The function *crackHTTPAuth* is selected in the right source code node, and the selected as possible case of re-use is marked on orange. The plain text representation of these two parts of source code shows that they are practically identical.

5 Conclusions and Future Work

The main goal of this research work is to provide a helpful tool for source code reviewers in order to help them to decide wheter or not a source code has been re-used. DeSoCoRe is the first online tool which it can detect source code re-use across languages as far of our knowledge.

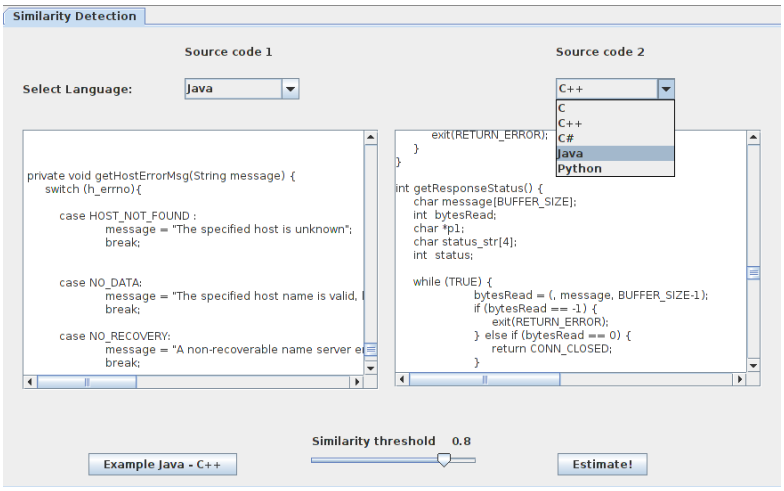
We have developed a methodology for detecting source code re-use across languages, and have shown their functionality by presenting DeSoCoRe tool, which works between and within programming languages. This makes our tool a valuable cross-lingual source code detector. DeSoCoRe allows comparing source codes written in Python, Java and C syntax family languages: C, C++ or C#. We plan in the next future to extend its functionality to other common programming languages. As future work we aim at allowing for the comparison at fragment level, where a fragment is considered a part of a function, a group of functions.

Acknowledgments

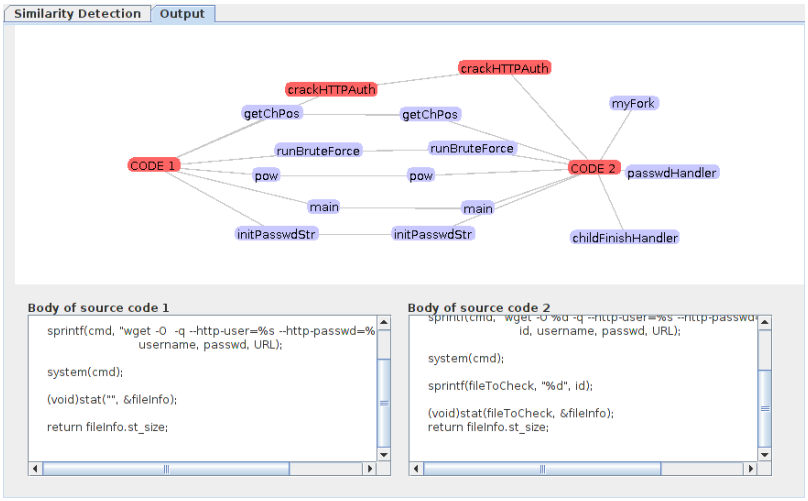
This work was done in the framework of the VLC/ CAMPUS Microcluster on Multimodal Interaction in Intelligent Systems and it has been partially funded by the European Commission as part of the WiQ-Ei IRSES project (grant no. 269180) within the FP 7 Marie Curie People Framework, and by MICINN as part of the Text-Enterprise 2.0 project (TIN2009-13391-C04-03) within the Plan I+D+i. The research work of the second author is funded by the CONACyT-Mexico 192021 grant.

[5] In agreement with (Flores et al., 2011), the default threshold for C-like languages (C, C++, Java...) is 0.8.

[6] Software tools for creating rich interactive data visualizations (*http://prefuse.org/*)



(a) Input screen: user have to select each language manually.



(b) Output screen: the re-used functions are connected using an edge and their codes are shown in the text areas below.

Figure 3: Screenshot of the interface of DeSoCoRe.

References

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

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

E. Flores, A. Barrón-Cedeño, P. Rosso and L. Moreno. 2011. Towards the Detection of Cross-Language Source Code Reuse. *Proceedings 16th International Conference on Applications of Natural Language to Information Systems, NLDB-2011, Springer-Verlag, LNCS(6716)*, pp. 250–253.

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

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

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

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

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