Towards the Code Clone Analysis in Heterogeneous Software Products

TIJANA VISLAVSKI, ZORAN BUDIMAC AND GORDANA RAKIĆ, University of Novi Sad

Code clones are parts of source code that were usually created by copy-paste activities, with some minor changes in terms of added and deleted lines, changes in variable names, types used etc. or no changes at all. Clones in code decrease overall quality of software product, since they directly decrease maintainability, increase fault-proneness and make changes harder. Numerous researches deal with clone analysis, propose categorizations and solutions, and many tools have been developed for source code clone detection. However, there are still open questions primarily regarding what are precise characteristics of code fragments that should be considered as clones. Furthermore, tools are primarily focused on clone detection for a specific language, or set of languages. In this paper, we propose a language-independent code clone analysis, introduced as part of SSQSA (Set of Software Quality Static Analyzers) platform, aimed to enable consistent static analysis of heterogeneous software products. We describe the first prototype of the clone detection tool and show that it successfully detects same algorithms implemented in different programming languages as clones, and thus brings us a step closer to the overall goals.

Categories and Subject Descriptors: **D.2.7 - [Software engineering - Distribution, Maintenance, and Enhancement]**: Restructuring, reverse engineering, and reengineering

Keywords: Software Quality, Software Maintainability, Static Analysis, Code Clone Detection

1. INTRODUCTION

Copy-paste activity is a common developer practice in everyday programming. However, this practice introduces code clones, parts of identical, or near-identical code fragments. It is estimated that between 5% and 23% of large-scale projects represents duplicated code [Roy et al. 2009] [Pulkkinen et al. 2015]. Such code is harder to maintain, increases potential errors and "bugs" and decreases overall quality of software [Dang and Wani 2015] [Sheneamer and Kalita 2016] [Roy et al. 2009] [Pulkkinen et al. 2015]. If original code has some error, by copy-paste activity this error is scattered on several places. Consequently, when this error is resolved later on, developer must pay attention to change the code in all of these places. Similarly, when some change or new functionality should be introduced in a part of code that is repeated in multiple places across the project, the same task is being repeated multiple times. Thus, we can conclude that code clones not only make the source code more complicated to maintain, but also increase the cost of maintenance.

"Identical or near identical source codes" [Sudhamani and Lalitha 2014], "segments of code that are similar according to some definition of similarity" [Rattan et al. 2013] and various other definitions across literature lack precision when defining code clones. Different authors define similarity in different ways and clones are described on different levels of granularity (ranging from sequences of source code to architectural clones). Common for all explanations is that clones come from copy-paste activity, with minor (or no) modifications. The point at which degree of modification becomes too big to consider two parts of code as clones is not clearly determined. Still, a generally adopted classification of code clones across literature exists [Dang and Wani 2015] [Sheneamer and

This work was partially supported by the Ministry of Education, Science, and Technological Development, Republic of Serbia, through project no. OI 174023.
Authors addresses: T. Vislavski, Z. Budimac, G. Rakić, University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics, Trg Dositeja Obradovica 4, 21000 Novi Sad, Serbia; email: tijana.vislavski@dmi.uns.ac.rs zjb@dmi.uns.ac.rs gordana.rakic@dmi.uns.ac.rs

In: Z. Budimac, Z. Horváth, T. Kozsik (eds.): Proceedings of the SQAMIA 2016: 5th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Budapest, Hungary, 29.-31.08.2016. Also published online by CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073)

Kalita 2016] [Sudhamani and Lalitha 2014] [Roy et al. 2009]. Clones are classified in four groups, as follows:

- Type-1: two code fragments are type-1 clones if they are identical, with exclusion of whitespace, comments and layout of code.
- Type-2: two code fragments are type-2 clones if they are identical, with exclusion of identifiers, types, whitespace, comments and layout of code.
- Type-3: two code fragments are type-3 clones if they are identical, with exclusion of some lines added, deleted or altered, identifiers, types, whitespace, comments and layout of code.
- Type-4: two code fragments are type-4 clones if they have the same behavior, but are syntactically different.

In their work, [Roy et al. 2009] introduced an example which illustrates each clone type, based on a simple example - function that calculates a sum and a product (Figure 1).

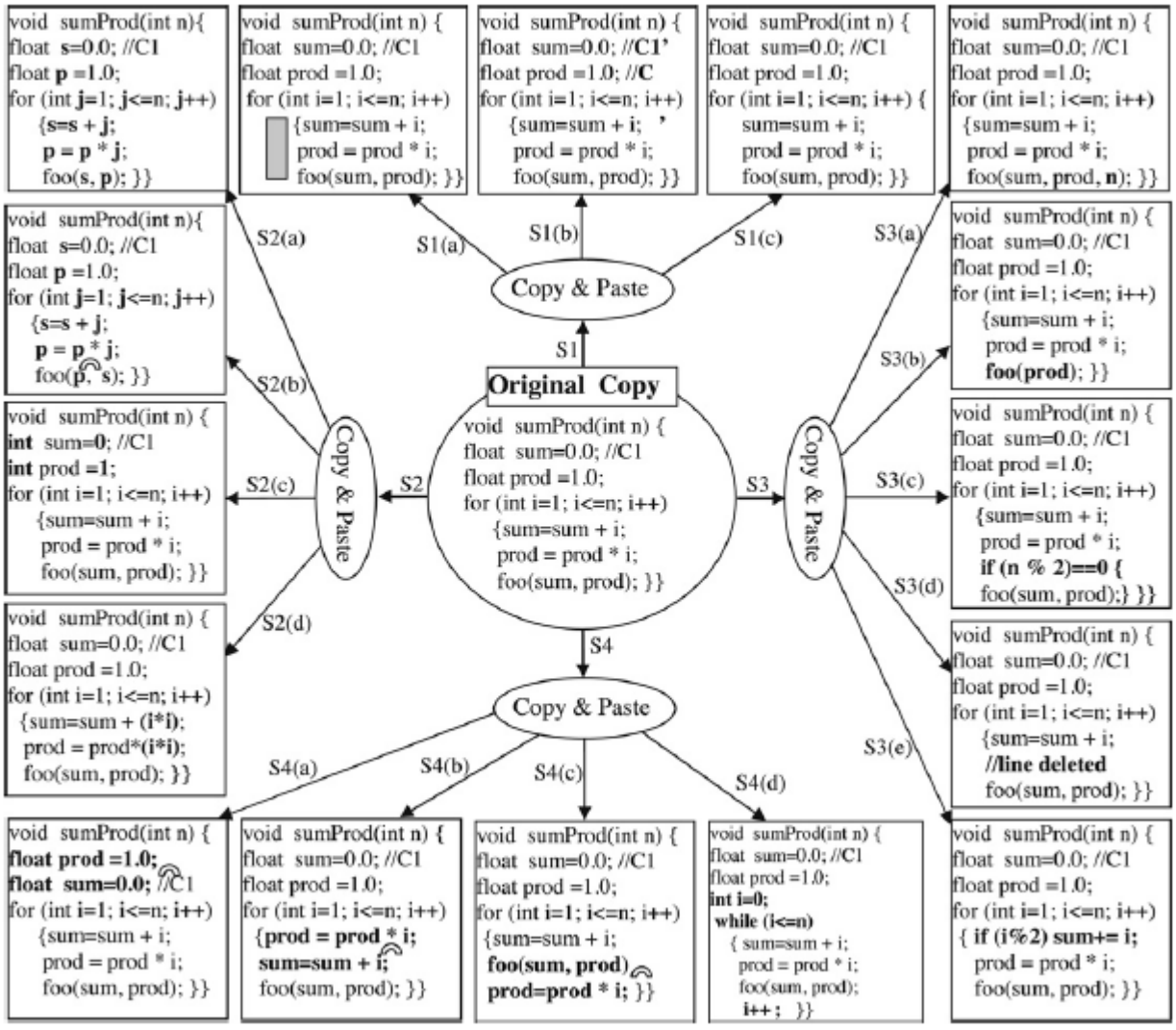

Fig. 1. Editing scenarios for each clone type [Roy et al. 2009]

When dealing with code clone detection, several possible approaches have been proposed [Sheneamer and Kalita 2016] [Roy et al. 2009] [Rattan et al. 2013]:

- Textual - compare two source code fragments based on their text, line by line, using none or little code transformations (such as removal of whitespace, comments etc.). These methods can be language-independent, but they mostly deal with Type-1 clones [Sheneamer and Kalita 2016] [Roy et al. 2009].
- Lexical - apply tokenization of source code, i.e. they transform source code in sequences of tokens. These approaches are generally more space and time consuming in comparison with textual [Sheneamer and Kalita 2016], but are more robust regarding some minor code changes, which textual approaches can be very sensitive to [Roy et al. 2009].
- Syntactical - either parse the source code into abstract syntax trees and then implement algorithms that detect matches in subtrees (tree-based), or they calculate several different metrics on parts of source code and compare results of these metrics. Metrics used are number of declaration statements, control statements, return statements, function calls etc. [Sheneamer and Kalita 2016] [Sudhamani and Lalitha 2014]
- Semantic - divided to graph-based and hybrid approaches. They are focused on detecting code parts that perform similar computation even when they are syntactically different. Graph-based approaches create PDGs (Program Dependency Graphs) that are then used to analyze data and control flow of different code parts. Hybrid approaches combine several different methods in order to overcome flaws that specific methods encounter [Sheneamer and Kalita 2016].

In the next chapter, we present some related work in this area. Chapter 3 describes language-independent source code representation in form of eCST (enriched Concrete Syntax Tree) and our algorithm for clone detection on that representation. Chapters 4 and 5 contain results and conclusions of our research, respectively. Finally, we propose some ideas for future work in Chapter 6.

2. RELATED WORK

Detailed description of various available tools has been given in [Dang and Wani 2015], [Sheneamer and Kalita 2016], [Roy et al. 2009], [Oliviera et al. 2015]. We will mention two that use similar approaches as ours.

In [Sudhamani and Lalitha 2014] and [Sudhamani and Lalitha 2015] a tool is presented that identifies clones by examining structure of control statements in code. They introduce a distance matrix which is populated with number of different control statements in two code fragments and also takes into account nested control statements. Similarity function is then used to calculate a similarity between these fragments, based on values from the matrix. Our tool also uses some kind of a distance matrix and then calculates similarity based on a similarity function. However, instead of taking into account only control statements, we compare by wider scope of language elements, including control statements (which are represented by LOOP_STATEMENT and BRANCH_STATEMENT universal nodes).

[Baxter et al. 1998] presents a tool for clone detection based on tree matching. Code is parsed into ASTs (Abstract Syntax Trees) and then subtree matching algorithms are used to detect identical or near identical subtrees. This is similar with our intermediary representation, but we abstract over the concrete language constructs and make the trees language-independent. In order to deal with scalability, they use a hashing function, so subtrees are first hashed into buckets and then only subtrees from the same bucket are examined. This is something we have to consider in future versions in order to achieve more scalable solution.

The most important thing that makes our tool different is that we aim at clone detection that is independent of language, but in a way that it detects code clones across compilation units written in

different programming languages. Great majority of today's large-scale software products are written in more than one programming language. Different components/modules are developed in different technologies, and thus there is a need for a tool which would overcome technology differences when detecting clones. There are tools that are language independent, especially some text-based tools, but they mostly deal with one language at a time. [Sheneamer and Kalita 2016] [Roy et al. 2009] Part of SSQSA called eCSTGenerator enables us to collect many files written possibly in different languages at the same time, and transform all of them into their respective eCSTs. Once eCSTs are generated, they can all be analyzed together, regardless of their original language.

3. DESCRIPTION

Our clone detection tool has been created as part of SSQSA (Set of Software Quality Static Analyzers) framework [Rakić 2015]. This project introduced its intermediary source code representation called eCST (enriched Concrete Syntax Tree) that consists of two types of nodes. Universal nodes represent language constructs on higher, more abstract level. Example of such nodes are COMPILATION_UNIT, FUNCTION_DECL, STATEMENT, etc. These nodes are language-independent and are internal nodes of the eCST. Analysis of universal nodes in eCSTs, generated from source codes in different languages, enables reasoning about the semantics of these source codes, even though they are written in different languages that can even use different programming paradigms. Leaves of eCSTs contain nodes that represent concrete syntactical constructs from the source code and are language-dependent. Examples of such nodes are ";", "for", ":=" etc.

In Figure 2 is presented a part of eCST generated for insertion sort algorithm. It is a subset of nodes generated for the whole compilation unit, with presented only characteristic universal nodes relevant for the analysis. The rest of the tree, especially leaf nodes are omitted in order to abstract over the concrete language of implementation. The highest level contains FUNCTION_DECL universal node which is used as a parent for all lower level nodes that capture the information about a sorting function. As we go down the tree, universal nodes begin to represent more specific language constructs, of finer granularity. For example, LOOP_STATEMENT is used to for capturing the information about any kind of loop and contains CONDITION node and sequence of nodes that represent statements, as its children nodes. These statements can be again some loops, branches, assignment statements, function calls etc. Mentioned insertion sort algorithm was developed in four different languages: Java, Modula-2, JavaScript and PHP. Figure 3 contains respective source codes.
Our clone detection algorithm implementation is based on working with universal nodes, and thus enables clone detection in functions that were written in different programming languages. We implemented a dynamic algorithm that compares two eCST subtrees representing functions by comparing their respective nodes. eCSTs are first scanned in order to extract only subtrees which represent function bodies. Each function body is then considered to be separate tree. These trees are transformed into sequences of nodes, using BFS (Breadth-First Search) strategy. Each two sequences are then compared by creating a matrix $m \times n$ (m and n being number of nodes in two sequences). Every pair of nodes (i, j), $0 \leq i < m, 0 \leq j < n$ is compared and the result is inserted into the matrix following dynamic programming principle:

$$matrix[i][j] = compare(sequence1[i], sequence2[j]) + min(matrix[i-1][j-1], matrix[i][j-1], matrix[i-1][j])$$

When matrix is full, algorithm searches for a "best match" between two trees, i.e. it searches a sequence of entries in the matrix going from $matrix[m-1][n-1]$ to $matrix[0][0]$ in which number of matched nodes is the greatest. Finally, similarity of two functions is measured with the formula

$(B / A) * 100\%$. B represents the number of matches in the best match and A is the total number of nodes on the path from $matrix[m-1][n-1]$ to $matrix[0][0]$ on which the best match was found.

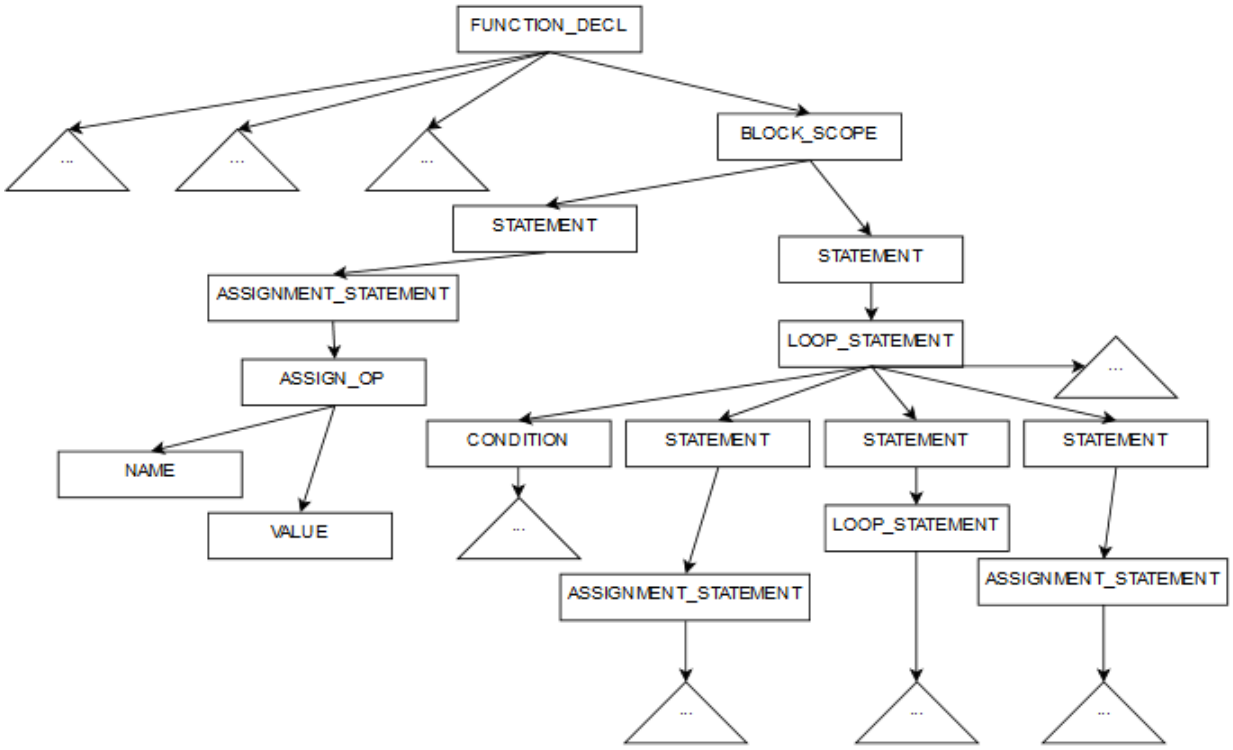

Fig. 1. Part of eCST generated for insertion sort algorithm

4. RESULTS

Algorithm was tested on three different sets of problems:

- scenario-based clones proposed in [Roy et al. 2009] and presented in Figure 1 in order to test our clone detection algorithm on one language and compare it with results of other tools
- implementation of few algorithms in Java, JavaScript, Modula-2 and PHP to check behavior of our tool on several different languages to test language-independence
- coding competition samples proposed in [Wagner et al. 2016] in order to test limitations of our algorithm and get some indications in which direction should we continue.

4.1 Algorithm correctness check

First we implemented scenario-based clones [Roy et al. 2009] in one language - Java. Since the nature of eCST is such that it ignores layout and whitespace (comments can be detected or ignored, it is configurable), Type-1 clones are easily detected with a 100% matching. All scenarios for Type-2 had a 100% matching, except of scenario *S2d* which had a 93% matching. This is because additional subtrees are generated for statements $sum = sum + (i \times i)$ and $prod = prod \times (i \times i)$, compared to $sum = sum + i$ and $prod = prod * i$ (for parts of statements in parenthesis) that do not have matches in original. And since total number of nodes for these functions are small (40 for original), few extra nodes make a big difference in overall calculation of similarity. Results for Type-3 ranged from 81% to 100%, and for Type-4 where all above 90%.

```java
public class InsertionSort {
    void sort(int[] array) {
        int i = 1;
        while (i < array.length) {
            int j = 0;
            while (j < i && array[j] < array[i]) {
                j++;
            }
            int temp = array[i];
            for (int k = i; k > j; k-- ) {
                array[k] = array[k-1];
            }
            array[j] = temp;
            i++;
        }
    }
}
```

```
IMPLEMENTATION MODULE InsertionSort;
    PROCEDURE Sort(array : ARRAY OF INTEGER);
    VAR i, j, k, temp : INTEGER;
    BEGIN
        i := 1;
        WHILE (i < array.Length) DO
            j := 0;
            WHILE (j < i) AND (array[j] < array[i]) DO
                INC(j);
            END;
            temp := array[i];
            FOR k:=i TO j - 1 BY -1 DO
                array[k] := array[k-1];
            END;
            array[j] := temp;
            INC(i)
        END
    END Sort;
END SelectionSort.
```

```javascript
function sort(array) {
    var i = 1;
    while (i < array.length) {
        var j = 0;
        while (j < i && array[j] < array[i]) {
            j++;
        }
        var temp = array[i];
        for (var k = i; k > j; k-- ) {
            array[k] = array[k-1];
        }
        array[j] = temp;
        i++;
    }
}
```

```php
function sort($array) {
    $i = 1;
    while ($i < count($array)) {
        $j = 0;
        while ($j < $i && $array[$j] < $array[$i]) {
            $j++;
        }
        $temp = $array[$i];
        for ($k = $i; $k > $j; $k-- ) {
            $array[$k] = $array[$k-1];
        }
        $array[$j] = $temp;
        $i++;
    }
}
```

Fig. 3. Insertion sort algorithm in Java, Modula-2, JavaScript and PHP

As we can see from the table, the biggest problem for our tool represented clones *S3c* and *S3e* which have whole additional branches inserted. Reason for this is in the structure of eCSTs created accordingly, similarly to scenario *S2d* explained above. New subtrees are generated for these branches that increase number of nodes in these two scenarios substantially (substantially in comparison with total amount of nodes that are created for the original function).

Since the paper which proposed clone scenarios [Roy et al. 2009] was comparing different tools based on their published characteristics, at this point we do not have empirical data to compare our detection of scenario-based clones with different tools. In [Roy et al. 2009] tools were categorized on the 7-level scale ranging from *Very well* to *Cannot* in terms of whether specific tool can detect certain scenario or not. Regarding the category of tree-based clone detection tools, to which our tool belongs to, authors determined that none of the tools from this category would be able to detect Type-4 clones, except *CloneDr* [Baxter et al. 1998] which would *probably* be able to identify scenario *S4a*. Following their scale range, we would place our first prototype in *Medium* level (3rd level), since we still do not have enough information about potential false positives, and we must for now presume that our tool may return substantial number of those.

Table I. Results for scenario-based clone example proposed by [Roy et al. 2009]

Type-2 clones				Type-3 clones					Type-4 clones			
S2a	S2b	S2c	S2d	S3a	S3b	S3c	S3d	S3e	S4a	S4b	S4c	S4d
100%	100%	100%	93.1%	100%	100%	81.53%	96.15%	83.33%	100%	96.15%	96.15%	92.72%

4.2 Cross-language consistency

In this phase two implementations of sorting algorithms, insertion sort and selection sort, as well as recursive function that calculates Fibonacci's numbers where considered. Implementations have been done in four different programming languages: Java, JavaScript, PHP and Modula-2. A part of eCST generated for insertion sort algorithm and respective source codes have already been given in Figures 2 and 3. These are semantically the same algorithms, only differences come from syntactic rules of their respective languages. Thus, slightly different trees are going to be generated. For example, Modula-2 function-level local variable declarations are located before the function block scope, so no VAR_DECL nodes are going to be presented in the BLOCK_SCOPE of Modula-2 function, in contrast to other languages.

4.3 Limitations

We used a sample of a dataset proposed in [Wagner et al. 2016], which represents various solutions to problems that were being solved at a coding competition. This set of problems was quite interesting since all implementations have a common goal - they solve the same problem. However, calculated similarities were quite small (not going over 30%), despite being written in the same language (Java). This corresponds to results published by [Wagner et al. 2016] where another class of clones is discussed - clones that were not created by copy-paste activity, but independently. These clones are called functionally similar clones (FSC). As in case of other tools [Wagner et al. 2016], ours was not able to identify this type of clones, and it is still an open issue to cope with.

5. CONCLUSION

With our clone detection algorithm we showed that even inter-language clones could be detected when operating on the level of universal nodes. Since most programming languages share the same concepts and similar language constructs, same algorithm implemented in two or more languages could produce the same eCST trees and thus their shared structure can be detected, which we showed on the few examples in Java, Modula-2, PHP and JavaScript. We also showed that our tool successfully identifies different copy-paste scenarios as highly similar code fragments. However, this is only the first prototype and has certain limitations and weaknesses. Our similarity calculation is very sensitive in respect of length of code. For example, when a substantial amount of code is added in between two parts of code that were result of a copy-paste activity, their similarity will decrease, perhaps even below some threshold we set up as a signal for clone pair, depending on the amount of code added.

6. FUTURE WORK

There is a lot of space for improvement in our tool, regarding current approaches and taking new ones. Our analysis is currently only dealing with function-level granularity. This should be extended in both ways - narrowing and widening it. Our similarity calculation is particularly sensitive to adding new parts of code or removing some parts (Type-3 clones), because it takes into account number of nodes which can change substantially with these changes. Our calculation should be

normalized in order not to fluctuate so drastically with these insertions and deletions. Also, since the algorithm compares all units of interest (currently function bodies) with each other, this is not a solution that would scale very good on large projects. A work-around should be carried out in order to deal with this problem, some grouping of similar units, either by using some sort of hash function, a metric value etc.

Regarding future directions, we could change our implementation to work not with eCSTs, but with eCFGs (enriched Control Flow Graphs) [Rakić 2015], which would allow us to concentrate more on semantics while detecting clone pairs and searching for architectural clones using eGDNs (enriched General Dependency Networks) [8], both representations already being part of SSQSA.

Output is currently only text-based, with calculated similarities for each two functions in some given scope, and optionally whole generated matrices. This kind of output could of course be improved, by introducing some graphical user interface which would, for example, color-map clone pairs in the original code.

REFERENCES

S. Dang, S. A. Wani, 2015. Performance Evaluation of Clone Detection Tools, *International Journal of Science and Research* Volume 4 Issue 4, April 2015

F. Su, J. Bell, G. Kaiser, 2016. Challenges in Behavioral Code Clone Detection, In *Proceedings of the 10th International Workshop on Software Clones*

A. Sheneamer, J. Kalita, 2016. A Survey of Software Clone Detection Techniques, *International Journal of Computer Applications* (0975 - 8887) Volume 137 - No.10, March 2016

M. Sudhamani, R. Lalitha, 2014. Structural similarity detection using structure of control statements, *International Conference on Information and Communication Technologies (ICICT 2014), Procedia Computer Science 46 (2015),* 892-899

C. K. Roy, J. R. Cordy, R. Koschke, 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computer Programming* 74 (2009), 470-495

P. Pulkkinen, J. Holvitie, O. S. Nevalainen, V. Lepännen, 2015. Reusability Based Program Clone Detection- Case Study on Large Scale Healthcare Software System, *International Conference on Computer Systems and Technologies - CompSysTech '15*

J. A. de Oliviera, E. M. Fernandes, E. Figueriedo, 2015. Evaluation of Duplicaded Code Detection Tools in Cross-project Context, *In Proceedings of the 3rd Workshop on Software Visualization, Evolution, and Maintenance* (VEM), 49-56

G. Rakić, 2015. Extendable And Adaptable Framework For Input Language Independent Static Analysis, Novi Sad, September 2015, Faculty of Sciences, University of Novi Sad, 242 p, doctoral dissertation

S. Wagner, A. Abdulkhaleq, I. Bogicevic, J. Ostberg, J. Ramadani, 2016. How are functionally similar code clones syntactically different? An empirical study and a benchmark, *PeerJ Computer Science* 2:e49 https://doi.org/10.7717/peerj-cs.49

D. Rattan, R. Bhatia, M. Singh, 2013. Software Clone Detection: A systematic review, *Information and Software Technology* 55 (2013), 1165-1199

M. Sudhamani, R. Lalitha, 2015. Duplicate Code Detection using Control Statements, *International Journal of Computer Applications Technology and Research* Volume 4 – Issue 10, 728 - 736

Baxter, A. Yahin, L. Moura, M. Anna, 1998. Clone detection using abstract syntax trees, *Proceedings of the 14th International Conference on Software Maintenance*, ICSM 1998, 368-377