Evaluating Clone Detection Techniques

Filip Van Rysselberghe
Lab On Re-Engineering
University Of Antwerp
Middelheimlaan 1, B 2020 Antwerpen
Filip.VanRysselberghe@ua.ac.be

Serge Demeyer
Lab On Re-Engineering
University Of Antwerp
Middelheimlaan 1, B 2020 Antwerpen
Serge.Demeyer@ua.ac.be

Abstract

In the last decade, several researchers have investigated techniques to detect duplicated code in programs exceeding hundreds of thousands lines of code. All of these techniques have known merits and deficiencies, but as of today, little is known on where to fit these techniques into the software maintenance process. This paper compares three representative detection techniques (simple line matching, parameterized matching, and metric fingerprints) by means of five small to medium cases and analyses the differences between the reported matches. Based on this experiment, we conclude that (1) simple line matching is best suited for a first crude overview of the duplicated code; (2) metric fingerprints work best in combination with a refactoring tool that is able to remove duplicated subroutines; (3) parameterized matching works best in combination with more fine-grained refactoring tools that work on the statement level.

1. Introduction

Code cloning or the act of copying code fragments and making minor, non–functional alterations, is a well-known problem for evolving software systems leading to duplicated code fragments or code clones. Of course, the normal functioning of the system is not affected, but without countermeasures by the maintenance team, further development may become prohibitively expensive [7, 18]. Fortunately, the problem has been studied intensively and several techniques to both detect and remove duplicated code have been proposed in the literature.

As far as removal of duplicated code is concerned, the state of the art proposes *refactoring* which is a technique to gradually improve the structure of (object-oriented) programs while preserving their external behaviour [17]. *Extract Method* which extracts portions of duplicated code in a separate method, is an example of a typical refactoring to remove duplicated code. However, quite often one must use a series of refactorings to actually remove duplicated code, as in *Transform Conditionals into Polymorphism* where duplicated conditional logic is refactored over the class hierarchy using polymorphism [7]. With refactoring tools like the refactoring browser [6] emerging from research laboratories into mainstream programming environments[1], refactoring is becoming a mature and widespread technique.

Concerning the detection of duplicated code, numerous techniques have been successfully applied on industrial systems. These techniques can be roughly classified into three categories. (i) *string-based*, i.e. the program is divided into a number of strings (typically lines) and these strings are compared against each other to find sequences of duplicated strings [8, 12]; (ii) *token-based*, i.e. a lexer tool divides the program into a stream of

[1] See http://www.refactoring.com/ for an overview of IDE's supporting refactoring

tokens and then searches for series of similar tokens [2, 13]; (iii) *parse–tree based*, i.e., after building a complete parse-tree one performs pattern matching on the tree to search for similar sub–trees [14, 15, 4]. On the first International Workshop on Detection of Software Clones, a number of research groups recently participated in a clone detection contest[2] to compare the accuracy of different tools against a benchmark of programs containing known duplication. The results of this experiment are currently being analysed by the participants.

Despite all this progress, little is known about the most optimal application of a given clone detection technique during the maintenance process. For instance, which technique should one use in a problem assessment phase, when one suspects duplicated code but isn't sure how much and in which files? Or which technique works best in combination with a refactoring tool, which has to know the exact boundaries of the code segment to be refactored, including possible renaming of variables and parameters? To answer these questions, this paper compares three representative *clone detection techniques* —namely simple line matching, parameterized matching, and metric fingerprints— by means of five small to medium cases. The reported matches as well as the process are analysed with special interest in differences. Afterwards, our findings are interpreted in the context of a generic software maintenance process and some suggestions are made on the most optimal application of a given technique.

The paper is structured as a comparative study, however due to the multiple aspects involved in the issue studied a more extensive experiment is necessary in the near future. A brief overview of existing duplicated code detection techniques is given in section 2. The experimental set-up, including the questions and cases driving the experiment are discussed in section 3. The results of section 4 are interpreted in section 5 to evaluate where the given technique might fit into the software maintenance process. Finally, section 6 summarises our findings in a conclusion.

2. Detection Techniques

The detection of code clones is a two phase process which consists of a *transformation* and a *comparison* phase. In the first phase, the source text is transformed into an internal format which allows the use of a more efficient comparison algorithm. During the succeeding comparison phase the actual matches are detected.

Due to its central role, it is reasonable to classify detection techniques according to their internal format. This section gives an overview of the different techniques available for each category while selecting a representative for each category.

2.1. String Based

String based techniques use basic string transformation and comparison algorithms which makes them independent of programming languages.

Techniques in this category differ in underlying string comparison algorithm. Comparing calculated signatures per line, is one possibility to identify for matching substrings [12]. Line matching, which comes in two variants, is an alternative which is selected as representative for this category because it uses general string manipulations.

Simple Line Matching is the first variant of line matching in which both detection phases are straightforward.

Only minor transformations using string manipulation operations, which can operate using no or very limited knowledge about possible language constructs, are applied. Typical transformations are the removal of empty lines and white spaces.

During comparison all lines are compared with each other using a string matching algorithm. This results in a large search space which is usually reduced using hashing buckets. Before comparing all the lines, they are hashed into one of *n* possible buckets. Afterwards all pairs in the same bucket are compared.

[2] http://www.informatik.uni-stuttgart.de/ifi/ps/clones/

Duploc is a Smalltalk tool which implements such a simple line matching technique [8], however also a Java version is available

Parameterized Line Matching is another variant of line matching which detects both identical as well as similar code fragments. The idea is that since identifier–names and literals are likely to change when cloning a code fragment, they can be considered as changeable *parameters*. Therefore, similar fragments which differ only in the naming of these parameters, are allowed.

To enable such parameterization, the set of transformations is extended with an additional transformation that replaces all identifiers and literals with one, common identifier symbol like "$". Due to this additional substitution, the comparison becomes independent of the parameters. Therefore no additional changes are necessary to the comparison algorithm itself.

Parameterized line matching is discussed in [9].

2.2. Token Based

Token based techniques use a more sophisticated transformation algorithm by constructing a token stream from the source code, hence require a lexer. The presence of such tokens makes it possible to use improved comparison algorithms.

Next to parameterized matching with suffix trees, which acts as representative, we include [13] in this category because it also transforms the source code in a token-structure which is afterwards matched. The latter tries to remove much more detail by summarising non interesting code fragments.

Parameterized Matching With Suffix Trees consists of three consecutive steps manipulating a suffix tree as internal representation.

In the first step, a lexical analyser passes over the source text transforming identifiers and literals in parameter symbols, while the typographical structure of each line is encoded in a non-parameter symbol. One symbol always refers to the same identifier, literal or structure. The result of this first step is a parameterized string or p-string.

Once the p-string is constructed, a criterion to decide whether two sequences in this p-string are a parameterized match or not is necessary. Two strings are a parameterized match if one can be transformed into the other by applying a one-to-one mapping renaming the parameter symbols. An additional encoding prev(S) of the parameter symbols helps us verifying this criterion. In this encoding, each first occurrence of a parameter symbol is replaced by a 0. All later occurrences are replaced by the distance since the previous occurrence of the same symbol. Thus, when two sequences have the same encoding, they are the same except for a systematic renaming of the parameter symbols.

After the lexical analysis, a data structure called a parameterized suffix tree (p-suffix tree) is built for the p-string. A p-suffix tree is a generalisation of the suffix tree data structure [16] which contains the prev()-encoding of every suffix of a P-string. Concatenating the labels of the arcs on the path from the root to the leaf yields the prev( )-encoding of one suffix. The use of a suffix tree allows a more efficient detection of maximal, parameterized matches.

All that is left for the last step, is to find maximal paths in the p-suffix tree that are longer than a predefined character length.

Parameterized matching using suffix trees was introduced in [2] with Dup as implementation example.

2.3. Parse-tree Based

Parse tree based techniques use a heavyweight transformation algorithm, i.e. the construction of a parse tree. Because of the richness of this structure, it is possible to try various comparison algorithms as well.

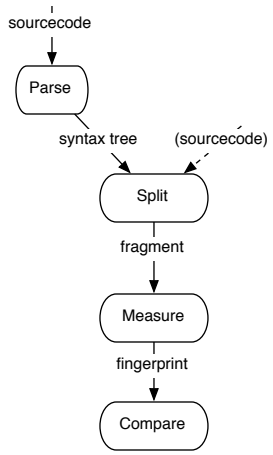

Figure 1. Detection steps for the metric fingerprint technique

The representing technique differs from [4] in that the latter uses sub–tree matching on the syntax tree.

Metric Fingerprints builds on the idea that you can characterise a code fragment using a set of numbers. These numbers are measurements which identify the functional structure of the fragment and sometimes the layout.

The metric fingerprint technique can be divided in five steps, each with a well-defined task. However the algorithm behind each task may differ between implementations. Figure 1 shows the basic steps in the detection process.

Before we can characterise the functional structure of a code fragment with numbers, it's wise to transform the source code into a representation that allows us to calculate such measurements efficiently. This transformation job is done using a *parser* which builds the syntax tree of the source code.

After parsing we end up with one large syntax tree. This tree is then *split* into interesting fragments. The choice of the type of fragments used is difficult because it affects the detection results. Most of the time, however, method and scope blocks are used as fragments since they are easily extracted from a syntax tree.

Afterwards the fragments are characterised through a set of measurements by *measuring* the values for a set of metrics, chosen in advance. This set of metrics can differ between various implementations, but most of the time it specifies functional properties. However there are implementations in which layout metrics are used as well. Cyclomatic complexity, function points, expression complexity (functional) and lines of code (layout) are examples of possible measures.

Finally, these sets of numbers are compared to each other. Depending on the implementation, algorithms with different levels of sophistication or power may be used. One possible approach calculates the Euclidean distance between each pair of fingerprints, considering fragments within zero distance as clones.

Both [14] and [15] describe a possible implementation of metric fingerprints. In the first, the metric set consists of 5 indirect metrics which are treated as a vector, while the latter uses 21 measures which are compared to each other using a system of hierarchical categories (an overview of both techniques can be found in [19]).

3. Research Approach

The research process used during our experiment is based on the Goal-Question-Metric paradigm which states that you should (1) outline a goal, (2) generate questions that verify whether the goal has been met and (3) select measures to answer them [3].

3.1. Goal

Identify which clone detection techniques are more appropriate for specific tasks of the maintenance process.

3.2. Questions

The questions we selected, were chosen because each highlights features that are of importance during the maintenance process. This way, these questions help us verifying whether our goal was accomplished.

Q1. How much configuration is needed to apply on another language? Before using a technique, you like to know how much configuration has to be done to adapt it to your particular programming context. Especially because it may limit the applicability of the technique, certainly in COBOL and C++ environments, where lots of dialects exist.

Q2. What kind of matches are found? Depending on the maintenance task at hand, you may be looking for specific kinds of duplication. For instance, during a problem assessment phase, maintainers want to obtain an overall report of the amount of duplication existing in all program files. On the other hand, during a restructuring phase, maintainers are interested in a duplication tool that detects only the programming constructs that one can restructure using a particular tool. Therefore a refactoring tool, moving methods in the class hierarchy, is interested only in duplicated method bodies.

Q3. How accurate are the results? For the clone detection problem, detection accuracy is difficult to define, but in the context of duplicated code detection it is characterised by three quality measures:

- number of false positives (to be minimised): that is, the number of matches the technique incorrectly identified as a piece of duplicated code.
- number of useless matches (to be minimised): that is, the number of matches which are not worth to be removed by means of refactoring. Typically depending on the length of a match.
- number of recognisable matches (to be maximised): that is, the number of matches that are easily recognised as interesting. For instance, in a program restructuring phase these are the matches that are easily removed by the refactoring tool at hand.

Q4. How does it perform? When using a detection technique one wishes to balance the amount of usable information that one can derive, with the time and memory one invested. Therefore you need to establish the performance of each technique and identify performance bottlenecks. This question addresses how the execution time of each technique relates to its input.

3.3. Experimental set-up

The next step after selecting research questions, consists of constructing an experiment that answers these questions. For the experiment reported in this paper following steps were conducted:

creation of reference implementations — Evaluating clone detection techniques differs from the evaluation of clone detection tools, in that it is the algorithm that is evaluated instead of the implementation. Differences in execution time between tools can for example be caused by the use of different programming languages or the application of techniques such as parallel computing. Unlike [5], which evaluates the results of various detection tools, this experiment focusses on the techniques themselves.

To evaluate each of these techniques, reference implementations of them were made in Java. Each of these implementations tried to adhere as closely as possible to the original technique's specification as given in [2]

for parameterized matching using suffix trees and [14, 15] for the metric fingerprint technique. For simple line matching such a reference implementation was already available and the original Duploc-tool[8] was used as an additional reference.

selection of cases — Five case were selected to evaluate the different techniques. These cases are representative for different degrees of duplication. Their limited size (under 10 000 LOC) allows an in-depth study of the duplication present as well as the reported matches. Section 3.4 describes each of the different cases.

application of the implementations — After selecting the cases, the different techniques were applied on each of them.

comparison and collection of results — At the end, the different matches were studied and compared with the different techniques. Data that was related to the execution of the different implementations like the execution time and its memory use, was studied as well.

3.4. Selected Cases

For the experiment, we selected five small to medium sized cases which are known to suffer from different kinds of duplication, although we did not know the exact locations of the duplicated code beforehand. Therefore, these cases are representative for various usage scenario's or different amounts of clones. Moreover, all cases are available on the web which allows replication of the experiment by other researchers studying duplicated code detection techniques. Following cases were used:

- ScoreMaster is a Java application automatically generated for the Enhydra web–server. Because most of the code has been generated automatically, it contains a high degree of duplication.
- TextEdit is an example project that is distributed with Borland's JBuilder to demonstrate GUI programming in Java. Due to its educational nature it contains little duplication[20].
- Brahms is music sequencing and notation software for linux written in C++ and was formerly known as KooBase. The small amount of duplication present is of a different nature because the code was written manually in an open source context[1].
- JMocha is a Java beans benchmark developed by IBM[11].
- JavaParser of JMetric is, as indicated by its name, a Java parser generated by Java for the JMetric project. It concerns a larger example of automatically generated code full of duplication[10].

4. Results

This section reports about the experiment by answering the questions listed under 3.2. A summary of these answers is given by table 1.

How much configuration is needed to apply on another language?

Simple line matching, as it only utilises basic string manipulations, is a truly language independent technique which is *very easy to configure*. As a language independent technique, no modification is required to be applicable on different languages.

All the remaining techniques on the other hand, do require configuration.

For parameterized matching the portability to another language is fair. Changing the lexer, which lies at the basis of both techniques, suffices to port it. Because more changes in the lexer are necessary for the parameterized

line matching technique, its portability is slightly lower than that of the suffix tree technique. Both parameterized techniques are *fairly portable*.

The metric fingerprint technique demands *much configuration effort as it is syntax dependent* due to the use of a parser. Even in our very first attempt to analyse a program, we were confronted with this syntax dependence because it failed due to a syntax error in the analysed code. The use of a parser limits the technique to syntactically correct sources of one language and makes changing to other languages difficult.

What kind of matches are found?

A rough classification of the clones found yields: *functional block duplication* and *general duplication*.

Functional block duplication characterises the duplication found by the metric fingerprint technique. Because this technique characterises functional blocks such as methods or code blocks by a fingerprint, only code fragments which share a functionally equivalent structure, are reported. The addition or removal of structures in a block violates this equivalence.

General duplication is found by the three other techniques. Everything that was duplicated, including pre–processor directives or comments, can be detected by them.

This last category can eventually be subdivided into the different fragments found by the corresponding techniques: *duplicated symbol blocks* for the suffix tree technique, *duplicated lines block* for parameterized line matching and *equal lines* for simple line matching. Duplicated in this context refers to the fact that parameter symbols may have changed.

How accurate are the results?

Number of false matches— *No false matches* are reported by both simple line matching and parameterized matching using suffix trees. Simple line matching reports only equal lines which makes it impossible to have false positives, while parameterized matching using suffix trees benefits from its P-string encoding that enforces a strict one–to–one parameterization. Only positive matches (parameterized or exact) are found by them

Parameterized line matching allows a non systematic renaming of the parameters which leads to *few false matches*. Such systematic renaming is necessary to ensure that two fragments share the same basis functionality which characterises duplication. Figure 2 shows an example, discovered in TextEdit. The problem especially seems to target GUI initialisation code. However reporting fragments consisting of a long sequence of matching lines instead of shorter ones, helps in keeping the number low. When we used this technique for ScoreMaster and Brahms we did not receive any false matches, while one false match was reported for TextEdit.

Even *more false matches* are reported by the metric fingerprint technique. Applying metric fingerprints with block-fragments resulted in over 200 false matches (cf. with 0 for the other 3 techniques) while only two were found using methods as fragments. The characterisation of expressions which lacks accuracy (see figure 3 for an example in ScoreMaster), is responsible for this problem. However it is our opinion that adding better expression metrics, like “expression complexity”, reduces this problem’s impact. Furthermore, less false matches are found when the granularity or size of the selected fragments is bigger. The number of false matches for this technique thus depends on the way expressions are characterised and the length of the fragments.

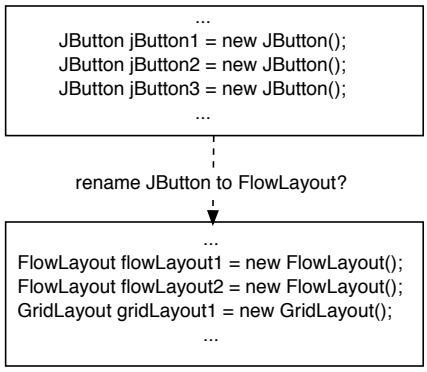

Figure 2. Example of a false match for parameterized line matching

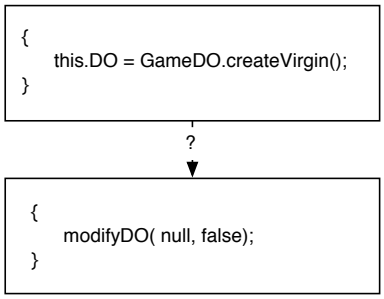

Figure 3. Example of a false match for metric fingerprints

Number of useless matches— The use of a threshold like in both parameterized matching techniques, keeps the number of useless matches *low*. Changing the threshold helped us in keeping the number of useless matches below 20.

For the metric fingerprint technique *more useless matches* are reported. Most of them are only one to four lines long and are caused because two method calls with the same number of arguments always match. For TextEdit for example, we found 133 useless matches on 138 reported matches (137 of them were valid matches) when we used method granularity. Using a threshold would reduce the amount of useless matches, especially in programs which contain many small methods or code blocks.

Simple line matching also reports *many useless matches*. For the same example as in the previous paragraph we got 229 useless matches. The problem here is that any program already contains some exactly matching lines by nature. As an example think of the “return;” statement you tend to write in your program. It is hard to estimate the exact number of useless matches in general but usually it is larger than the amount for metric fingerprints.

Number of recognisable matches— For the metric fingerprints technique the number is *high*. Each match that is returned is a *functional block* like e.g. scope blocks and method definitions.

Both parameterized matching techniques return a *lower* number of recognisable matches. It is difficult to decide which matches are important by just looking at the output because each match represents a chunk of duplicated lines or symbols, which lacks context.

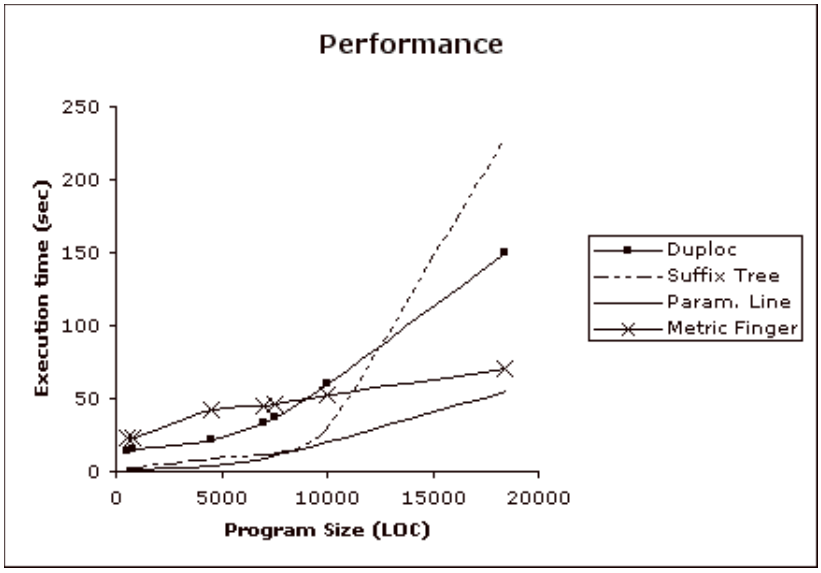

Figure 4. Performance of the different techniques

The number of recognisable matches for simple line matching is *even lower* (reduced from 4 with parameterized matching to 2). All exactly matching lines are reported. Visualisation can be used to detect the interesting duplicates. However the lack of parameterization makes it more difficult than the parameterized techniques to detect altered duplicates.

How does it perform?

Because the actual performance of a technique depends on many factors like implementation and testing platform, we started by calculating the theoretical time complexities. For both line matching techniques this results in a time complexity of O(n^2) because each line is compared with each other line resulting in an exponential complexity. Using Ω hash buckets as proposed in [8] reduces this complexity to O($\frac{n^2}{\Omega}$). Parameterized matching using suffix trees on the other hand, has a complexity of O($|\Pi| * n$) (with $|\Pi|$ the number of parameter symbols) as was formally proven by Baker in [2]. As a last technique we studied the time complexity of the metric fingerprint technique which shows a time complexity of O(m^2) when a simple comparison is used to compare the *m* fragments.

Afterwards we compared these complexity formulas with the execution times we measured[3] leading to a couple of rather interesting observations. A first observation was the problem of page swapping. From a certain point (in our experiment 10 000 LOC) the linearity of the suffix tree could no longer be maintained. The reason for this was the *page swapping* which was necessary to store the whole suffix tree in memory. Memory space is thus a constraining factor when analysing large projects.

A second observation was the unexpectedly high performance of the parameterized line matching technique. The execution time of this technique showed a very flat exponential tendency. Better memory use and shorter comparisons due to shorter strings, are reasons for that performance.

Figure 4 shows how the execution time for each technique relates to the input size. It clearly shows our two observations as well as an overview of each technique's performance.

[3]Testing platform was a pentium 200Mhz with 64MByte RAM

	Portability	Duplication	Matches: number of			Scalability
			False	Useless	Recognisable	
Simple Line	$+++++$	general lines		$----$	$+$	$+$
Param. Line	$+++$	general line block	$-$		$++++$	$+++$
Suffix Tree	$++$	general token block			$++++$	$++$
Metric. Fing.		functional entity	$--(--)$	$--(--)$	$+++++$	$+++$

Table 1. Summary of the relation between each technique and the properties studied. The number of symbols indicates the comparative degree of satisfaction of the property studied. Positive properties are marked with +, negative with -. The additional symbols placed between brackets, denote the additional impact when using block–granularity

5. Interpretation

A first observation we made, was the difference in scalability of the various techniques. By applying each technique on a common case, we were able to get in touch with the scalability of the different techniques, something we could not derive from the theoretical time complexities alone. Who could ever imagine that the relative execution time of parameterized line matching increases much slower than its simple counterpart while an additional transformation is applied?

During our experiment we were certainly puzzled by the major difference in execution time (2 minutes versus 8) for the suffix tree technique when advancing from 7500 LOC to 10710 LOC, certainly because a linear time complexity was formally proven for this technique. As analysis of the memory showed, page swapping was the reason for this behaviour. By experimenting we found that parameterized matching using suffix trees has problems sustaining its theoretical linearity due to memory restrictions which in turn limits its scalability.

For the comparison of the output of the techniques, we also used a visualisation tool. Quite often this visual comparison showed striking differences in the outputs. At one moment for example, we were really stunned by the large amount of matches reported when we used block–fragments instead of method–fragments in the metric fingerprint technique. However our amazement was of short notice because investigation of the various fragments revealed a large number of false and useless matches. Comparison with other techniques supported this idea immediately. Using block-granularity for metric fingerprints did not only cost much more time and memory, but also resulted in a large amount of useless information.

After this we immediately compared the method-granularity with the remaining techniques. The number of matches drew our immediate attention as metric fingerprints finds a number of very small (1 or 2 lines), yet useless matches. However, the remaining large matches were duplicated methods, which usually are easy to refactor. The limited amount of matches combined with their clear content makes the technique useful in a first, coarse refactoring phase.

At first sight, the parameterized techniques and simple line matching seemed to report different duplicates, while the difference in output between our two parameterized techniques was small. However, a second more in-depth look at the reports revealed that sometimes very small matches were found by simple line matching while the parameterized techniques found an entire fragment. A small amount of duplicates was not even found by simple line matching because in each line at least one parameter symbol was altered. This indicates that some very detailed duplication was missed. Applying parameterized matching resulted in more detailed and more recognisable matches.

6. Conclusion

In this paper we have studied three duplicated code detection techniques, which are representative for the techniques published in the literature. By means of five small to medium cases (some of them including generated code, hence having lots of duplication) we compared the results, focussing on those portions where the techniques performed differently. Based on this experiment, we make the following conclusions.

- *Simple line matching* (representative for the string-based techniques) gives a crude overview of the duplicated code that is quite easy to obtain, hence is most appropriate during problem detection and problem assessment.
- *Parameterized matching* (representative for the token-based approaches) provides a precise picture of a given piece of duplicated code and is robust against rename operations. Therefore it works best in combination with fine-grained refactoring tools that work on the level of statements (i.e. *Extract Method*, *Move Behaviour Close to Data*, and *Transform Conditionals into Polymorphism*.
- *Metric fingerprints* (representative for the parse-tree based techniques) are very good at revealing duplicated subroutines, irrespective of small differences, hence work best in combination with refactoring tools that work on the method level (i.e. *Remove Method* and *Pull up method*);

These results are preliminary in nature and should be confirmed by other experiments. First of all, future experiments should incorporate large and very-large (over a million lines of code) programs into the set of cases to see whether our results still hold. Secondly, the same experiment should be done with other techniques to see whether our findings indeed generalise across the given categories.

Despite these limitations, we have shown that the different clone detection techniques reported in the literature each have specific advantages compared to the others. As such, each technique is more appropriate for a certain maintenance task. In that sense, this paper laid the foundation for a more systematic way of detecting and removing duplicated code.

7. Acknowledgements

We would like to thank Gerd Van Den Heuvel, whose master's thesis provided the necessary means for conducting the experiments described in this paper. We also would like to thank Stéphane Ducasse, Bart Du Bois and Andy Zaidman for reviewing the paper. Matthias Rieger was helpful by providing us with an implementation of Duploc.

References

[1] Brahms. *http://brahms.sourceforge.net*. by Sourceforge.

[2] B. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering 1995*, 1995.

[3] V. R. Basili and H. D. Rombach. The tame project: Towards improvement–oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758 – 773, 1988.

[4] I. Baxter, A. Yahin, L. Moura, and M. S. Anna. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, 1998.

[5] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Second IEEE International Workshop on Source Code Analysis and Manipulation(SCAM '02)*, October 2002.

[6] J. B. D. Roberts and R. E. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253 – 263, 1997.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt, 2002.
[8] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance*, 1999.
[9] G. V. D. Heuvel. Parameterized matching: a technique for the detection of duplicated code. Master's thesis, University of Antwerp, 2002.
[10] Jmetric. *http://www.it.swin.edu.au/projects/jmetric/products/jmetric*. by School of Information Technologie at Swinburne University of Technology.
[11] Jmocha. *http://www–124.ibm.com/developerworks/opensource/jmocha/*. by IBM.
[12] J. Johnson. Identifying redundancy in source code using fingerprints. In *Cascon*, 1993.
[13] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Engineering*, 28(7):654 – 670, 2002.
[14] K. Kontogiannis, R. Demori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1), 1996.
[15] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance*, 1996.
[16] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 32(2):262–272, 1976.
[17] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
[18] D. Parnas. Software aging. In *Proceedings of The 16th International Conference on Software Engineering*, 1994.
[19] F. V. Rysselberghe. Detecting duplicated code using metric fingerprints. Master's thesis, University of Antwerp, 2002.
[20] Textedit. *http://www.borland.com/jbuilder*. by Borland.