Comparison of Clone Detection Techniques

Technical Report 2012-593

Saeed Shafieian
QRST Group, School of Computing
Queen's University, Kingston, Canada
saeed@cs.queensu.ca

Ying Zou
Department of Electrical and Computer Engineering
Queen's University, Kingston, Canada
ying.zou@queensu.ca

Abstract— **Many techniques for detecting duplicated source code (software clones) have been proposed in the software reengineering literature. However, comparison of these techniques in terms of performance is not widely studied. There are four general categories for clone detection techniques; textual, lexical, syntactic, and semantic. This report presents an experiment that evaluates different clone detectors based on four Java programs of small to medium size scales. These subject systems have been used in the recent literature, and can be considered as standard systems for this purpose. At least one clone detection tool has been tested for each category. The comparison of different techniques is done based on performance metrics for clone detection tools. The most widely used metrics, precision and recall, have been used to calculate quantitative values for the performance of different techniques so that they can be compared with each other. The reference clones used in the comparison are those in the Bellon corpus. Our goal was to only evaluate systems that were not previously evaluated using Bellon benchmark, and not to replicate the previous works in our main experiment.**

Keywords- software clones; clone detction techniques; clone detection tools; comparison.

I. INTRODUCTION

Reusing code fragments by copying and pasting with or without minor modifications is a common activity in software development. As a result, software systems often contain sections of code that are very similar, called code clones. There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, which is called textual similarity, or they can be similar based on their functionality (independent of their text), which is called functional similarity. There are four clone types in total, in which the first three are textual and the last one is functional [1][3]. In the first three groups, the degree of textual similarity between clone pairs/group decrease, so for example Type 3 clones are less similar to each other than Type 1 clones are. This also means that detecting Type 3 clones is harder than detecting Type 2 or Type 1. Detecting Type 4 clones, which are also called semantic clones, is undecidable in general [2], so we have only focused on detecting Type 1, Type 2, Type 3 clones in this project.

Several studies show that software systems with code clones are more difficult to maintain than the ones without them [10][14]. The tendency of cloning not only produces code that is difficult to maintain, but may also introduce difficult to detect errors. Code clones are considered as one of the bad smells of a software system and it is widely believed that cloned code has several adverse affects on the maintenance life cycles of software systems. Therefore, it is beneficial to remove clones and prevent their introduction by constantly monitoring the source code during its evolution.

Code cloning is found to be a more serious problem in industrial software systems [4][12][14][17]. In presence of clones, the normal functionality of the system may not be affected, but without countermeasures by the maintenance team, further development may become prohibitively expensive [23]. Code clones may adversely affect the software systems' quality, especially their maintainability and comprehensibility. For example, if a bug is found in a code fragment, all of its similar cloned fragments should be detected to fix the bug in question. Moreover, too much cloning increases the system size and often indicates design problems such as missing inheritance or missing procedural abstraction [12]. Although the cost of maintaining clones over a system's lifetime has not been estimated yet, it is at least agreed that the financial impact on maintenance is very high. Grubb estimates the costs of changes carried out after delivery at 40% - 70% of the total costs during a system's lifetime. Existing research shows that a significant amount of code of a software system is cloned code and this amount may vary depending on the domain and origin of the software system [24].

Clone detection tools can be compared based on different criteria. For example, they can be compared according to usage (platform, external dependencies, availability), according to interaction (user interface, nature of output, IDE support), or based on language (language paradigm, language support) [1]. However, in this project we will compare different tools according to Bellon clone evaluation benchmark [2]. The performance of different techniques will be quantitatively and qualitatively compared with each other. For this purpose, precision and recall metrics will be calculated for each technique, and used as performance evaluation metrics. The rest of the report is organized as follows. Section II gives an overview of the related works done for comparing and evaluating clone detection tools and

techniques. Section III discusses our proposed approach in comparing different clone detection techniques, and comparing them to Bellon results. Some of the tools used in this experiment are new, and were not used by Bellon. Section IV shows the results achieved using the mentioned approach. In Section V some of the problems that we faced are mentioned. Finally, Section VI concludes the report, and illustrates the future work.

II. RELATED WORK

One of the first experiments done for comparing clone detection tools was conducted by Burd and Bailey [5]. They compared three state-of-the-art clone detection and two plagiarism detection tools. They began by validating all the clone candidates of the subject application obtained with all the techniques of their experiment to form a human oracle, which was then used to compare the different techniques in terms of several metrics to measure various aspects of the reported clones. Although they were able to verify all the clone candidates, the limitations of the case study in terms of a single subject system, a graph layout tool developed in 1999 at the University Of Durham, modest system size and validation subjectivity may make their findings not comprehensive. Moreover, the intention of their analysis was to assist in preventative maintenance tasks, which may have influenced their clone validation process.

Considering the limitations of Burd and Bailey's study, Bellon et al. set out to conduct a larger tool comparison experiment [2] on the same three clone detection tools used in Burd and Bailey's study and three additional clone detection tools. They also used a more diverse set of larger software systems, consisting of four Java and four C systems totaling almost 850 KLOC. As in the study of Burd and Bailey, a human oracle validated a random sample of about 2% of the candidate clones from all the tools evenly and blindly. While their study is the most extensive to date, only a small proportion of the clone candidates were oracled and several other factors may have influenced the results [25]. Bellon's framework has been reused in experiments by Koschke et al. [6][7], Ducasse et al. [8] (partially), and Selim et al. [18].

Rysselberghe and Demeyer [9][23] have evaluated prototypes of three representative clone detection techniques, providing comparative results in terms of portability, kinds of duplication reported, scalability, number of false matches, and number of useless matches. However, they did not make a reference set, used relatively small subject systems (under 10 KLOC) and did not provide the reliability of the oracle(s) that validated the detected clones. Moreover, rather than quantitative evaluation of the detection techniques, their intention was to determine the suitability of the clone detection techniques for a particular maintenance task (refactoring) which might have influenced their clone validation.

Roy and Cordy [1][3] have performed one of the most comprehensive studies in comparing and evaluating clone detection tools and techniques. They provide a qualitative comparison and evaluation of the to date state-of-the-art in clone detection techniques and tools, and organize the large amount of information into a coherent conceptual framework. They classify, compare and evaluate the techniques and tools in two different dimensions. First, they classify and compare approaches based on a number of facets, each of which has a set of (possibly overlapping) attributes. Second, they qualitatively evaluate the classified techniques and tools with respect to taxonomy of editing scenarios designed to model the creation of Type-1, Type-2, Type-3 and Type-4 clones. They also provide examples of how one might use the results of this study to choose the most appropriate clone detection tool or technique in the context of a particular set of goals and constraints.

One of the latest studies in this area was done by Selim et al. [18]. They presented a hybrid clone detection technique, which complements string or token-based clone detectors to detect Type 3 clones by leveraging the intermediate representation. They used systems from the Bellon benchmark and through a manual quantitative and qualitative evaluation, showed that their technique is able to detect Type 3 clones. The recall rates for their technique were higher than those for source-based clone detectors with minimal drop in the precision using Bellon corpus, which has incomplete clone groups. Their technique also has slightly higher precision than the standalone string and token-based clone detectors.

III. OVERVIEW OF APPROACH

We have used three Java applications from the Bellon benchmark, and one other Java application from apache project as our subject and test systems. The reason for choosing these systems as subject systems is to enable me to compare our results to those of Bellon et al [2]. Furthermore, Bellon benchmark has been used in the recent literature in other papers like [18]; therefore we will be able to compare the results with their results, too.

There are four major categories for clone detection techniques, which are textual or text-based techniques, lexical or token-based techniques, syntactic techniques, and semantic techniques [1][3]. Syntactic approached can be further divided into tree matching approaches and metrics-based approaches [1]. Some authors consider these last two approaches as different approaches, however we will use the general classification performed by Cordy in this report.

For each of the four major categories for clone detection techniques, we have used at least one subject system, except for the last one, as the representative of that category. The availability of clone detectors in different categories varies, for example there are some good text-based and token-based clone detectors available, whereas finding a detector for the semantic approach is really difficult. The reason for this is that most of the tools are academic tools, and were mainly developed for the purpose of the published papers based on

them. As a consequence, a downloadable and working version does not exist for all of them.

The Bellon benchmark is an experimental setup suggested by Bellon et al. [2] as a means of standardizing the evaluation of clone detectors. In this benchmark, due to the time needed to manually verify the results, only 2% of the clone groups reported by the clone detectors are randomly selected and evaluated by Bellon. Evaluation is done incrementally where 1% of the reported clone groups are 'oracled' for evaluation, and then another 1% is tested. Clone groups validated by Bellon as correctly identified clone groups are used to build a reference corpus. Each reported clone group is referred to as a 'candidate', and each correctly identified clone group is referred to as a 'reference'. Further details of the original setup are provided in [2].

Now, we will explain the four major categories for clone detection approaches, and the tools we have used for each category.

A. Textual Approaches

Textual approaches (or text-based techniques) use little or no transformation/normalization on the source code before the actual comparison, and in most cases raw source code is used directly in the clone detection process [1].

There are a few clone detectors that find clones based on similarity in code strings. These types of clone detection tools are generally only able to find Type 1 clones [18]. For the purpose of this study, we have used Simian [19], NiCad [27][21], and SDD [28][22] as clone detectors for the textual category.

Simian (Similarity Analyzer) identifies duplication in Java, C#, C, C++, COBOL, Ruby, JSP, ASP, HTML, XML, Visual Basic, Groovy source code and even plain text files. It runs natively in any .NET 1.1 or higher supported environment and on any Java 5 or higher virtual machine. In this project, we have used the Java version of Simian, which is a command line tool. It can generate outputs in plain text and XML formats.

NiCad is a flexible TXL-based hybrid language-sensitive, text comparison software clone detection developed by James R. Cordy and Chanchal K. Roy based on Chanchal's PhD thesis work. It uses syntactic pretty-printing with flexible code normalization and filtering, then textual comparison with thresholds. It provides output results in both XML form for easy analysis and HTML form for convenient browsing. NiCad is currently available for installation as a command-line tool on Linux, Mac OS X and Cygwin. To run NiCad on a system, FreeTXL compiler/interpreter must first be installed on that system. NiCad runs in two modes, functions and blocks. Functions mode only detects function clones, and blocks mode only detects blocks of code. Therefore, it cannot detect sequence of codes as code clones when they are neither functions nor blocks. This makes comparing this tool with other ones difficult. However, we decided to evaluate it in this project because it was not evaluated using Bellon benchmark before.

SDD is a clone detection tool that can be used as an Eclipse plugin. It uses data structure of an inverted index and an index with n-neighbor distance concept. It has three properties, which can be configured based on the specific subject system; pattern for file matching: a pattern of the target files for SDD that determines which types of files should be used for clone detection, N-neighbor length: suitable values are 2 or 3, and minimum chain size: the minimum size of similar parts, for which normally 15 or more is good. In this project, we have used 2 for n-neighbor length and 15 for minimum chain size.

Comparing the results SDD clone detector with the Bellon reference corpus was very difficult, because this tool generates the results graphically in an Eclipse window, and there is not any file associated with them.

As stated before, our goal in this study is to evaluate the tools that have not been evaluated before using Bellon reference corpus. Therefore, we will not use Simian for the main experiment, and will only use NiCad and SDD for this purpose.

B. Lexical Approaches

Lexical approaches (or token-based techniques) begin by transforming the source code into a sequence of lexical 'tokens' using compiler-style lexical analysis. The sequence is then scanned for duplicated subsequences of tokens and the corresponding original code is returned as clones. Lexical approaches are generally more robust over minor code changes such as formatting, spacing, and renaming than textual techniques [1].

For lexical or token-based approaches, we have used PMD's CPD [20], which is able to detect clone Types 2 as well as Type 1. This tool can be used as an Eclipse plugin, and generates reports in plain text format, XML and in CSVs. It gets the minimum number of tokens to be detected for each clone pair as an input. In this project the default value of 25 was used. However, 25 tokens can build even a two-line code, which is not an appropriate clone size according to Bellon benchmark. Therefore, clone pairs/groups, which consisted of less than six lines, were removed from the results manually.

The other clone detector that we have used for the category of token-based tools is iClones [26]. iClones is an incremental clone detection tool that can extract clone evolution data from a program's history. For this purpose, it needs to have access to the source codes of different versions of the program. It generates an RCF (Rich Clone Format) file that contains clone evolution data. This file can then be analyzed using another tool called Cyclone or RCFViewer. However, since the goal of this project is not analyzing clone evolutions among different versions of a subject system, we have used iClones in the single-version mode, which like other clone detectors, operates on a single version of a subject system.

C. Syntactic Approaches

Syntactic approaches use a parser to convert source programs into parse trees or abstract syntax trees (ASTs), which can then be processed using either tree matching or

structural metrics to find clones [1][3]. In tree matching approaches, or tree-based techniques, clones are detected by finding similar sub-trees. Variable names, literal values and other leaves (tokens) in the source may be abstracted in the tree representation, allowing for more sophisticated detection of clones. On the other hand, metrics-based approaches gather a number of metrics for code fragments and then compare metrics vectors rather than code or ASTs directly [1][3].

For this category of clone detectors, there are even more limited tools available than the previous two categories, especially for the metrics-based subcategory. We have tried CloneDR [17] in our test experiment, and CloneDigger [29] for the main experiment.

In CloneDr, a compiler generator is used to generate an annotated parse tree (AST) and compares its sub-trees by characterization metrics based on a hash function. Source code of similar sub-trees is then returned as clones. The hash function enables one to do parameterized matching and to detect gapped clones, especially if the gaps are within a line. Unlike other clone detection tools, CloneDr is a commercial product, and the trial version only reports 10 sample clones of medium size (max 50 lines). Therefore, it could not be used for the main run, in which we need full results for calculating precision and recall.

CloneDigger uses XML representation of ASTs and anti-unification and code abstraction to find software clones in the source code. This tool uses adapters, which convert source files into an XML representation of their abstract syntax trees. Currently there are adapters for two languages: Python and Java. Adapters for other languages can be created, e.g. by using parser generators or using internal compiler representations. It takes several threshold values, including minimum clone size, as input parameters. It then produces a nice HTML file with a list of clones. Each pair is reported statement by statement with a highlighting of differences.

D. Semantic Approaches

Semantics-aware approaches have also been proposed, using static program analysis to provide more precise information than simply syntactic similarity. In some of these approaches, the program is represented as a program dependency graph (PDG). The nodes of this graph represent expressions and statements, while the edges represent control and data dependencies. This representation abstracts from the lexical order in which expressions and statements occur to the extent that they are semantically independent. The problem of finding clones is then turned into the problem of finding isomorphic sub-graphs [1][3].

There are very few clone detection tools available for this category. Therefore, in this project we did not evaluate any semantic-aware clone detectors.

E. Performance Evaluation

We compare the performance of clone detection tools with each other. Our goal is to find out which clone detection technique performs better than the others, at least based on the subject systems used in this project. We measure the performance using Recall and Precision, which are calculated as shown in equations (1) and (2). Precision is the number of reference clone groups detected by each technique relative to all the candidate clone groups detected by that technique. Recall is the number of reference clone groups detected by each technique relative to all of the reference clone groups available in the Bellon benchmark for that specific system.

$$Precision = \frac{|Detected \quad References|}{|Candidates|} \tag{1}$$

$$Recall = \frac{|Detected \quad References|}{|References|} \tag{2}$$

IV. CASE STUDY

This section describes our experiment with clone detection tools. The experiment is divided into two phases; the test run and the main run.

A. Test Run

The goal of the test run was to identify potential problems for the main run. The test phase analyzed three Java programs, EIRC, Suple, and Apache Ant. Eteria IRC Client (EIRC), is an Internet Relay Chat client program written in Java. Spule, Secure Practical Universal Lecture Evaluator, is a Java application that automates the evaluation of lecture polls. And the last one, Apache Ant, is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Table I shows the characteristics of these systems.

In the test run, it was noticed that some tools report the start and end lines of the code fragments a line earlier or later if the lines consist of only a brace. In practice, this difference is irrelevant, but it complicates the comparison of clones from different tools. For this reason, the source code for the main run need to be normalized, such that blank lines are removed and lines containing only opening or closing braces are removed and the braces are added to the line above. Therefore, we used a normalized Java application from Bellon benchmark for the main run.

The results of the test run on the previously mentioned applications are shown in Tables II, III, IV, and V. Clone detection tools used in the test run were Simian, NiCad, PMD's CPD, and CloneDr. Therefore, two text-based, one token-based, and one tree-based clone detectors were used in the test experiment.

Table II shows an overview of the results found by Simian for the three test systems. The parameters used for Simian were

TABLE I
OVERVIEW OF TEST SUBJECT SYSTEMS

System	KLOC in Java	# Java Files
EIRC	12	65
Spule	13	58
Apache Ant	254	1176

TABLE II
CLONE DETECTION RESULTS IN THE TEST RUN USING SIMIAN

System	# Clone pairs/groups detected	Max # of Lines	Clone Types
EIRC	29	23	Type 1
Spule	46	19	Type 1
Apache Ant	435	94	Type 1

its default parameters. This tool reports the detected clones as clone groups. The output generated in XML format was used during the test run. As the table shows, this tool was only able to detect Type 1 or exact clones. Clones less than six lines were removed from the results and not counted.

NiCad operates in two modes: functions and blocks. With the functions granularity, only function clones are detected, and with blocks granularity, clones that make a block are found. As a consequence, NiCad does not detect sequences of statements as code clones. This makes the comparison of this tool with other ones difficult. However, since it is a new tool, and generated good results in the run, it was used for the main run too. Table III shows the results obtained from running NiCad on the three test systems. As the table shows, this tool was able to detect all the three types of software clones.

NiCad reports results with the thresholds of 0%, 10%, 20%, and 30%. The results shown in Table III are for the blocks granularity and the threshold 30%, because these are the most comprehensive results, and other thresholds are also covered. This tool reports the found clones in clone groups. Clones less than six lines were removed from the results and not counted. Table IV illustrates the results of clone detection by CPD for the test systems. CPD was unable to detect clones of Apache Ant due to lexical error. This tool was able to find clones of Type 1 and Type 2.

The clone detection tool tested in the category of syntactic tools was CloneDr. Table V shows the results obtained using this commercial tool for finding clones in the test systems. Like the previous tool, CloneDr was not able to find clones in Apache Ant due to parsing error. The similarity threshold used for the test run was 95%. As Table V shows, this tool was able to detect Type 1 and Type 2 clones for EIRC, and only Type 1 clones for Spule. Clones of the sizes less than six were removed, and not counted in the results.

TABLE III
CLONE DETECTION RESULTS IN THE TEST RUN USING NICAD

System	# Clone pairs/groups Detected	Max # of Lines	Clone Types
EIRC	40	18	Type 1, Type 2, Type 3
Spule	46	46	Type 1, Type 2, Type 3
Apache ant	727	148	Type 1, Type 2, Type 3

TABLE IV
CLONE DETECTION RESULTS IN THE TEST RUN USING CPD

System	# Clone pairs/groups detected	Max # of Lines	Clone Types
EIRC	91	44	Type 1, Type 2
Spule	202	24	Type 1, Type 2

TABLE V
CLONE DETECTION RESULTS IN THE TEST RUN USING CLONEDR

System	# Clone pairs/groups detected	Max # of Lines	Clone Types
EIRC	50	46	Type 1, Type 2
Spule	63	47	Type 1

B. Main Run

Bases on the test run, and problem we realized during that phase, we selected the following clone detection tools: NiCad and SDD as text-based clone detectors, iClones and CPD as token-based clone detectors, and CloneDigger as the tree-based clone detector. The goal of this phase is to compare the performance of these tools according to precision and recall metrics.

We have chosen Netbeans Javadoc as our subject system. Therefore, the performance of these five clone detection tools will be compared based on clones they find in the subject system.

As mentioned before, we have used Bellon reference corpus for the purpose of comparison. We have also used the normalized version of the source code for the subject system as used by Bellon et al. In this version of Netbeans Javadoc, all blank lines have been removed, and lines containing only curly braces are adjusted. Table VI shows an overview of the subject system used for the main experiment. Two of the tools were tested for the first time in the main run, iClones and CloneDigger. Table VII shows an overview of the clone detection results for these tools. For iClones, the default configurations were used. As the Table VII shows, this tool was able to detect all the three clone types in the subject system. This tool is able to generate the output in RCF format, which makes the comparison process simpler.

TABLE VI
OVERVIEW OF THE SUBJECT SYSTEM FOR THE MAIN RUN

System	# Java Files	# Comment Lines	# Code Lines	Total KLOC in Java
Netbeans Javadoc	101	4780	9580	14

TABLE VII
OVERVIEW OF RESUTLS FOR NETBEANS-JAVADOC USIING ICLONES AND CLONEDIGGER

Clone Detector	# Clone pairs/groups detected	Max # of Lines	Clone Types
iClones	43	110	Type 1, Type 2, Type 3
CloneDigger	680	54	Type 1, Type 2

For the CloneDigger tool, only one parameter was changed during the experiment; size-threshold. This parameter determines the minimum clone size to be detected, so when this parameter is used, there is no need to removes clones less than six lines manually. CloneDigger was considerably slower than the other tools, but was able to find considerably much more clones pairs than the other tools. It can operate in a fast mode, which is faster, but finds fewer clones. As Table VII shows, this tool was able to find clones of Type 1 and Type 2 only. In fact, it can find clones only if they have exactly the same size. This tool was more successful in finding the first two type clones than the other tools.

The most important part in the main run is evaluating the performance of the discussed clone detection tools. For this purpose, precision and recall metrics values are calculated based on the Bellon reference corpus for the subject system. The parameters used for the clone detectors are the same as those used in the test run. Table VIII shows the performance results of the tools used for the main experiment.

The number of reference clones pairs found was very similar for the four text-based and token-based tools. However, the tree-based tool, CloneDigger, found considerably more clone pairs. As a consequence, the recall value for the first four tools are similar, but it is higher for the tree-based tool. The low precision value for the tree-based tool is because of the fact that it found much more clone pairs than the other tools. Therefore, for this tool there is a trade-off between precision and recall.

Figure 1 and Figure 2 show the Precision and Recall values for the five tested clone detection tools in comparison with each other. As Fig. 1 shows, iClones has the highest Precision value and CloneDigger has the lowest one among all the five tested clone detection tools. However, as Fig. 2 shows, CloneDigger has the highest Recall value and iClones has the lowest one among all the tested clone detectors. Therefore, it seems that there is a trade-off between high Precision and high Recall values for these tools.

TABLE VIII
PERFORMANCE OF TOOLS IN THE MAIN RUN FOR NETBEANS-JAVADOC

Category	Clone Detector	Precision	Recall
Text-Based	NiCad	0.17	0.22
	SDD	0.24	0.22
Token-Based	iClones	0.26	0.20
	CPD	0.11	0.24
Tree-Based	CloneDigger	0.03	0.36

V. PROBLEMS

One of the first problems we faced during this study was finding working clone detection tools. In fact, there are many clone detection techniques based on the software reengineering literature, however the real developed tools for all of these techniques are not available online, or at least are very hard to find. The other issue in this regard was setting up the tools. These trivial problems took a lot of time during the whole period of the experiments.

The other issue was finding the appropriate reference clone pairs. We were only able to find reference clones for Netbeans Javadoc application, which was used in the main run. We tested another reference clone set, which was for EIRC application, but it did not match any of the three available versions of EIRC. The other reference sets were very large, and could not be used in the time frame we had for this study.

The last problem was the ambiguity in setting the right parameters for the tools. These settings can greatly affect the number and quality of clones found by some of the tools. As there was no standard for this purpose, we used the default values as far as possible.

VI. CONCLUSION AND FUTURE WORK

Four Java applications ranging from 12 to 254 KLOC were used as subject systems in this study. Clone detection was done on these systems using five clone detectors from three different categories. Each tool represents a different technique, since even all the tools in the same category do not use the same algorithm for finding software clones. In order to compare the quality of different clone detection tools and techniques, we used Precision and Recall as the performance evaluation metrics.

The text-based and the token-based tools had very similar recall values, but their Precisions were different due to the fact that they found different numbers of clone pairs. Nevertheless, the tree-based tool had a noticeably higher recall value, and a noticeably less precision value than the tools in the other categories. All of these tools reported clones as clone groups, except for CloneDigger that reported results in clone pairs.

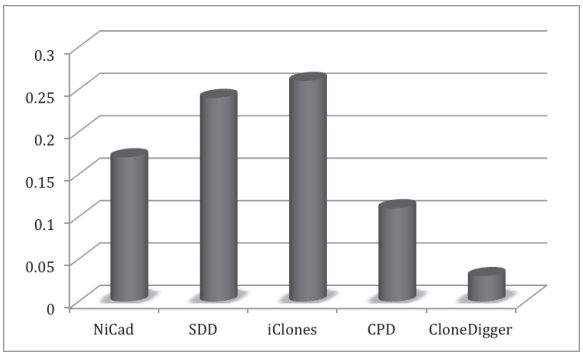

Fig. 1. Precision values of the different techniques

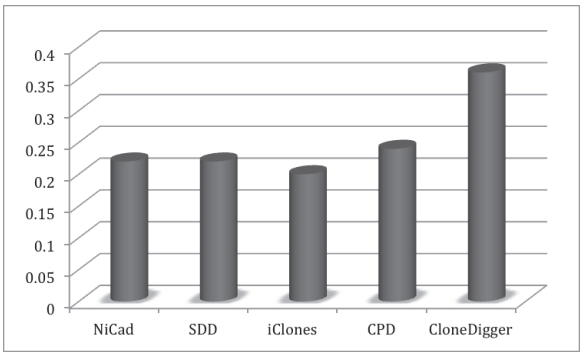

Fig. 2. Recall values of the different techniques

The results were graphical only for the SDD tool. For the other tools, the results were mainly in plain text, html, or CSVs.

All the empirical results reported in this report were done using a Mac OS X Lion machine, except for CloneDigger and CloneDr, which were tested on a Windows machine due to the lack of Mac OS versions. All the comparisons done in this study were performed manually, which was very time consuming. One of the future works may be writing a program to automatically compare candidate clones with reference clones, and calculates precision and recall values. This will greatly help in reducing the time required in performing the tests, and will enable us to focus more on other aspects of the study.

One of the other future works may be including the evaluation of semantic-aware clone detection tools based on Bellon reference corpus. The other one could be automating the comparison and evaluation process, so that more tools can be compared in a certain amount of time. The other improvement could be including other benchmark metrics other than quality metrics for comparing the results. The other improvement can be involving more subject systems in the evaluation process.

ACKNOWLEDGMENT

We would like to thank Dr. Rainer Koschke for providing us with the iClones tool. We also thank Peter Bulychev for providing us with the latest version of CloneDigger. The first author thanks Dr. Cordy for answering his inquiries regarding NiCad.

REFERENCES

[1] Roy CK, Cordy JR, Koschke R. Comparison and evaluation of clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 2009; 74:470–495.

[2] Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, Transactions on Software Engineering 33 (9) (2007) 577–591.

[3] Roy CK, Cordy JR. A survey on software clone detection research. *TR 2007-541,* Queen's School of Computing, 2007; 115.

[4] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A multi linguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654–670.

[5] E. Burd, J. Bailey, Evaluating clone detection tools for use during preventative maintenance, in: Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2002, 2002, pp. 36–43.

[6] R. Koschke, R. Falke, P. Frenzel, Clone detection using abstract syntax suffix trees, in: Proceedings of the 13th Working Conference on Reverse Engineering, WCRE 2006, 2006, pp. 253–262.

[7] R. Falke, R. Koschke, P. Frenzel, Empirical evaluation of clone detection using syntax suffix trees, Empirical Software Engineering 13(2008)601–643.

[8] S. Ducasse, O. Nierstrasz, M. Rieger, On the effectiveness of clone detection by string matching, International Journal on Software Maintenance and Evolution: Research and Practice 18 (1) (2006) 37–58.

[9] F. Rysselberghe, S. Demeyer, Evaluating clone detection techniques from a refactoring perspective, in: Proceedings of the 9th IEEE International Conference Automated Software Engineering, ASE 2004, 2004, pp. 336–339.

[10] J. Johnson, Identifying redundancy in source code using fingerprints, in: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1993, 1993, pp. 171–183.

[11] J. Johnson, Visualizing textual redundancy in legacy source, in: Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative research, CASCON 2004, 1994, pp. 171–183.

[12] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, ICSM 1999, 1999, pp. 109–118.

[13] B. Baker, A program for identifying duplicated code, in: Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, vol. 24, 1992, pp. 49–57.

[14] B. Baker, On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, 1995, pp. 86–95.

[15] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposiumon Static Analysis, SAS 2001, 2001, pp. 40–56.

[16] J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001, 2001, pp. 301–309.

[17] I. Baxter, A. Yahin, L. Moura, M. Anna, Clone detection using abstract syntax trees, in: Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, 1998, pp. 368–377.

[18] G. M. K. Selim, K. C. Foo and Y. Zou. "Enhancing Source-Based Clone Detection Using Intermediate Representation", Proc. WCRE, 2010, pp. 227-236.

[19] Simian Website [Online]. Available: http://www.harukizaemon.com/simian/, last accessed in December 2011.

[20] PMD's CPD Website [Online]. Last Accessed November 2011. URL http://pmd.sourceforge.net/cpd.html.

[21] NiCad Website [Online]. Last Accessed December 2011. URL http://www.txl.ca/nicaddownload.html.

[22] SDD Website [Online]. Last Accessed December 2011. URL http://wiki.eclipse.org/Duplicated_code_detection_tool_(SDD).

[23] Filip Van Rysselberghe, Serge Demeyer. Evaluating Clone Detection Techniques. In Proceedings of the International Workshop on Evolution of Large Scale Industrial Applications (ELISA'03), 12pp., Amsterdam, The Netherlands, September 2003.

[24] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. In IEEE Transactions on Software Engineering, Vol. 32(3): 176-192, March 2006.

[25] B. Baker, Finding clones with dup: Analysis of an experiment, IEEE Transactions on Software Engineering 33 (9) (2007) 608–621.

[26] iClones Website [Online]. Last Accessed December 2011. URL http://softwareclones.org/iclones.php.

[27] C. K. Roy, J. R. Cordy, NICAD:Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in: Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, 2008, pp. 172–181.

[28] S. Lee, I. Jeong, SDD: High performance code clone detection system for large scale source code, in: Proceedings of the Object Oriented Programming Systems Languages and Applications Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA Companion 2005, 2005, pp. 140–141.

[29] P. Bulychev, M. Minea, Duplicate code detection using anti-unification, in: Spring Young Researchers Colloquiumon Software Engineering, SYRCoSE 2008, 2008, p. 4.