Evaluating Clone Detection Tools for Use during Preventative Maintenance

Elizabeth Burd, John Bailey

The Research Institute in Software Evolution
University of Durham
South Road
Durham, DH1 3LE, UK

Abstract

This paper describes the results of a process whereby the detection capability of 5 code replication detection tools for large software application are evaluated. Specifically this work focuses on the benefits of identification for preventative maintenance that is with the aim to remove some of the identified clones from the source code. A number of requirements are therefore identified upon which the tools are evaluated. The results of the analysis processes show that each tool has its own strengths and weaknesses and no single tool is able to identify all clones within the code. The paper proposes that it may be possible to use a combination of tools to perform the analysis process providing that adequate means of efficiently identifying false matches is found.

1. Introduction

Software systems provide vital support for the smooth running of an organisation's business. It is the responsibility of maintainers to keep the system up-to-date and functioning correctly. It is reported by Burd [Bur97] that "*during the maintenance of legacy code, it is common to identify areas of replicated code*". These duplicate or near duplicate functionalities are termed clones [Lag97]. Within this paper a clone is recognized to be where a second or more occurrences of source code is repeated with or without minor modifications. Software cloning because of its ad hoc nature it is not considered reuse, quite the opposite, Mayrand argues that the need for such cloning indicates an organisation "*does not have a good re-use process in place*" [May97].

From the analysis of software application it appears that the inclusion of these clones results from the addition of some extra functionality which is similar but not identical to some existing logic within a system. Its seems that when presented with the challenge of adding new functionality the natural instinct of a programmer is to copy, paste and modify the existing code to meet the new requirements and thus creating a software clone [Bur97]. While the basis behind such an approach is uncertain, one possible reason is due to time restrictions on maintainers to complete the maintenance change. Duccase [Duc99] points out that "*making a code fragment is simpler and faster than writing from scratch*" and that if a programmer's pay is related to the amount of code they produce then the proliferation of software clones will continue.

It is generally recognised that the majority of effort during maintenance is classified as perfective changes i.e. "*expanding the existing requirements of a system*" [Tak96]. Software cloning complicates the maintenance process by giving the maintainers unnecessary code to examine.

Once a clone is created it is effectively lost within the source code and so both clones must therefore be maintained as separate units despite their similarities. If errors are identified within one clone then it is likely that modifications may be necessary to the other counter-part clones [Kom01]. Detection is therefore required if any of the clones are to be re-identified to assist the maintenance process. Further potential also exists for clone detection to assist the maintenance process. If clones can be detected then the similarities can be exploited and replaced during preventative maintenance with a new single code unit this will eliminate the problems identified above.

One major problem in detecting a clone is that it is impossible to be absolutely certain that one section of code has been copied and pasted from another. Short sections of code like wrapper methods in Java can have an almost identical structure to countless others. Furthermore, there are two types of cloning identified in [Bur97] "*Replication Within Programs*" describing situations where code has been copied and pasted once or more within the same file. Secondly "*Replication*

Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)
0-7695-1793-5/02 $17.00 © 2002 **IEEE**

Tool	Author	Supported Languages	Domain	Approach Category	Background
CCFinder	T.Kamiya	C, C++, COBOL, Java, Emacs Lisp, Plain Text	Clone Detection	Transformation followed by token matching	Academic
CloneDr	I. Baxter	C, C++, COBOL, Java, Progress	Clone Detection	Abstract Syntax Tree comparison	Commercial
Covet	J. Bailey J. Mayrand	Java	Clone Detection	Comparison of Function Metrics	Academic
JPlag	G. Malpohl	C, C++, Java, Scheme	Plagiarism Detection	Transformation followed by token matching	Academic
Moss	A. Aiken	Ada, C, C++, Java, Lisp, ML, Pascal, Scheme	Plagiarism Detection	Unpublished	Academic

Table 1: clone detection tools under investigation

Across Programs" is introduced as the cloning of code between files. Duccase [Duc97] defines "*cloned files*" as "*files that have a very high duplication ratio between each other"*.

Cloned code can constitute a significant proportion of a legacy system's code. Estimates regarding the scale of this problem vary depending on the domain and origin of the source code. For instance, Baxter [Bax98] has identified up to 12.7% of code being clones; Baker [Bak95] has identified between 13% - 20% of code being cloned and Lague [Lag97] between 6.4% - 7.5%. Taking these points into account it follows that any reduction of redundant code is beneficial to the maintenance of a system.

However, the problem is not solely restricted to the issue of the increasing size of an application. When code is copied and pasted systematic renaming of variables can lead to "*unintended aliasing, resulting in latent bugs*" [Joh94]. Thus, Johnson establishes that cloning is a form of "*software ageing*" or making even minor changes to the system's design very difficult.

There are a good number of clone detection tools available both commercially and within academia. Within these tools several different approaches to software clone detection have been implemented, including string analysis, program slicing, metric analysis and abstract tree comparisons. The aim of this paper is to compare a set of clone detection tools on some large software applications. The results of the analysis process will then be used to investigate which of the tools are best suited to assist the process of software maintenance in general and specifically preventative maintenance. The investigation will examine results gained from each tool using two metrics; that of precision and recall. Also of interest is investigating how similar the results achieved are in detecting replication within a single program and replication across distinct programs. Of further interest is how similar the results from the different categories of detection tools for example JPlag and Moss will only detect replication across programs because they are searching for cheating and copying and modifying one's own code is not plagiarism.

The following section will describe some of the existing tools in the field of clone detection.

2. Clone Detection Technique

Five established detection tools will be used in the evaluation process; JPlag, MOSS, Covet, CCFinder and CloneDr. JPlag and MOSS are web-based academic tools for detecting plagiarism in student's source code. CloneDr and CCFinder are stand alone tools looking at code duplication in general. Also included in the experiment is a prototype tool created Covet. Covet uses metrics by Mayrand [May96] and compares the metrics of each function to look for potential clones.

Table 1 summarizes the clone detection tools under investigation. The languages supported by the analysis process are highlighted, as is the analysis approach. The column labeled domain highlights the main purpose of the tools for either clone detection or for plagiarism detection.

Further details of the different approaches taken to examining source code by the clone detection techniques are now given.

CCFinder [Kam01] focuses on analyzing large-scale systems with a limited amount of language dependence. It transforms the source code into tokens. CCFinder aims to identify "portions of interest (but syntactically not exactly identical structures)". After the string is tokenised a token-by-token matching algorithms is performed. CCFinder also provides a dotplotting visualisation tool that allows visual recognition of matches within large amounts of code.

CloneDr [Bax98] analyses software at the syntactic level to produce abstract syntax tree (AST) representations. A series of algorithms are then applied to the tree to detect clones. The first algorithm searches for sub-tree matches within the ASTs. Then a "sequence detection" algorithm attempts to detect "variable size sequences of sub-tree clones". A third algorithm uses combinations of previously detected clones and looks for "more complex near-miss clones". The final clone set includes the clones detected in the second and third algorithms. CloneDr can automatically replace cloned code by producing a functionally equivalent subroutine or macro.

Covet uses a number of the metrics as defined by Mayrand [May96]. These metrics were selected by taking known clones and identifying which of the Datrix metrics best highlighted the known clone set. Covet does not apply the same scale of clone likelihood classification as Mayrand. Rather within Covet this is simplified; there is no scale of clone, functions are either classed as clones or distinct. The tool is still in the prototype stages and is not capable of processing industrial sized programs.

JPlag [Pre00] uses tokenised substring matching to determine similarity in source code. Its specific purpose is to detect plagiarism within academic institutions. Firstly the source code is translated into tokens (this requires a language dependent process). JPlag aims to tokenise in such way that the "essence" of a program is captured and so can be effective for catching copied functionality. Once converted the tokenised strings are compared to detect the percentage of matching tokens which is used as a similarity value. JPlag is an online service freely available to academia.

MOSS [Aik02] Aiken does not publish the method MOSS uses to detect source code plagiarism, as its ability to detect plagiarism may be compromised. Moss like JPlag is an online service provided freely for academic use. Source code is submitted via a perl script and then the results are posted on the MOSS's webpage. Users are emailed a url of the results.

3. Approach

If different approaches to clone detection are taken then the results achieved will vary considerably. How much variation is to be addressed by comparing the results of several clone detection tools. Both commercial and non-commercial tools are used in the experiment, including two that use clone detection techniques in order to find plagiarism in academia.

The experiment will involve running a set of clone detection tools over source code of GraphTool. GraphTool is a graph layout tool developed in 1999 at the University Of Durham. It is used internally within the computer science department. GraphTool was written in Java by a postgraduate and currently consists of 63 individual source files, 16335 lines of code totally 660KB. It was chosen because it is a medium size application. Also GraphTool is written in Java and so will be "understood" by the majority of detection software.

Each tool produces a set of matches (clone relationships) these results will be analyzed to assess the similarity between the resulting sets. In order to compare the results of each tool some translation will be required to allow comparison of intersection and difference of the result sets. To standardize the results of the different tool and to perform a comparison the start and end lines from each function is taken as the indices. This is necessary due to the different approach utilized by the different tools, for instance, Covet looks for cloned functions rather than disparate segments of code. Whereas the other tools provide start and end line indices showing exactly where the clones appear, this is not possible with Covet.

In order to establish the coverage of each tool their output was translated into a single data structure, a GeneralPair. This GeneralPair holds two matched sections of code (either within the same file or from separate source files). Implemented in Java it holds the name of the file(s) involved and the code regions that have been matched. Each code region is identified by a start and end index (line numbers). Each tool's GeneralPairs are stored in a set. If two GeneralPairs overlap then it is considered a match. For example

FileA.java(110-130)	&	FileB.java(340-360)	GeneralPair_One
FileB.java(310-350)	&	FileA.java(115-150)	GeneralPair_Two

GeneralPair_One and GeneralPair_Two are considered as a match because the code region in both FileA and FileB overlap.

An initial comparison of the tools is made through the use of two metrics that of Precision and Recall. Precision is the number of clones of the identified set that are also in the clone base. Thus precision is the measure to the extent to which the clones identified by the tool are extraneous or irrelevant. Recall is the number of clones in the clone base that are also in the identified set. Recall is therefore a measure to the extent to which the clones identified by the tool matches the clone base within the GraphTool application. In order to establish the clone base (the total number of clones within GraphTool) the results of clone identification from all tools were merged and then manually verified. This verification process is currently still subjective but the rejection of clones has been based on their unsuitability to assist the preventative maintenance process. Below is a description of the attributes considered during verification.
GraphTool was originally developed by a single postgraduate. As a result GraphTool's source code is not overly large with consistent naming conventions and formating. This consistency allowed familiarity with the code to develop fairly quickly and spotting replication easier and thus helped verification

Similar / Identical control flow and layout Series of of repeated layout blocks could often to point to a copy of another piece of code elsewhere in the system. For example if two functions both contained the same number of if-statements testing similar conditions.

Similar / Identical method names These usually took the form of a verb-noun combination with the verb remaining constant and the noun being changed. (for example saveGraph and saveGINGraph)

Similar / Identical variables Clusters of indentical variables and assignments were often a good indication that the code originated somewhere else.

Similar / Identical comments Occasionally the same or ver similar comment blocks were interspersed in the code. This is quite obviously a legacy of cloning within the source.

4. Results

In total the GraphTool case study identified 1463 initial clones. Not all clones were found by each tool, in fact no single tool identified all clones. The initial clone numbers identified by each tool are shown within Table 2 below.

	CCFinder	CloneDr	Covet	JPlag	Moss
Identified clones by tool	1128	84	278	131	120

Table 2: Total clone numbers identified by each tool

From Table 2 it can be seen that CCFinder identified the largest number of clones, a total of 1128, whereas CloneDr identified the smallest number only 84.

In order to examine the differences between the output of the two tools the difference between the sets of clones identified has been established. These are represented within Table 3.

	CCFinder	CloneDr	Covet	JPlag	Moss
CCFinder		1090	1089	989	1025
CloneDr	43		265	120	111
Covet	251	70		120	109
JPlag	44	73	273		67
Moss	19	76	268	81	

Table 3: The difference between the set of clones output between each tool

Thus Table 3 shows that CCFinder identified 1090 clones that were not identified by CloneDr, however CloneDr identified a further 43 that were not identified by CCFinder.

In addition the intersection between each of the tools has also been established. The intersections, i.e. those clones that were identified by more than one tool, are shown in Table 4. Thus, CCFinder and CloneDr identified 38 clones in common.

	CloneDr	Covet	JPlag	Moss
CCFinder	38	27	87	101
CloneDr		13	11	9
Covet			15	10
JPlag				50

Table 4: The intersection of identified clones from each tool

Due to the different domains of the detection tools that have been analyzed it is interesting to investigate for the set of all clones what proportion are identified within a single file and the proportion of clones that are identified across files within the GraphTool application. Due to the nature of their implementation, plagiarism detectors do not investigate clones within files, since this is not plagiarism. It is therefore of interest to see what potential disadvantages to clone identification for maintenance such restrictions would bring. The results of this analysis are represented within Table 5.

	Within files	Across files
CCFinder	44	66
CloneDr	79	21
Covet	31	69
JPlag	N/A	100
Moss	N/A	100

Table 5: The percentage of identified clones identified within / across files

The results show that CCFinder identifies the greatest proportion of its clones across files; that is CCFinder implies that most clones appears to be copied between a number of application files. However, the results of CloneDr seems to show the opposite in that the clones that it identifies are mostly copied within files. These results show that the plagorism detection software, due to only investigating replication across files, failed to identify over 500 clones.

What conducting the analysis process on only the total numbers of clones identified may obscure, is the overall proportion of the code that is cloned. For instance, the large numbers gained could be due to the relatively small size of the clones found. Therefore to gain an indication of the proportion of clones, mean size of the clone set has been analyzed for each tool. The results of this analysis process are shown within Table 6.

	Largest identified clone (LOC)	Mean size of clone (LOC)
CCFinder	94	17
CloneDr	100	16
Covet	123	21
JPlag	78	23
Moss	57	15

Table 6: The Lines of Code (LOC) of clones identified

Table 6 shows some interesting results. Is is surprising that the mean size of the clones that are identified do not significantly vary for each tool, with the exception of Moss where both the maximum and mean sizes are slightly smaller than the others.

A further interesting aspect of the clone identification process is the number of times each clone is replicated within the code. The data presented so far identifies the numbers of total clones; that is all duplicate instances of the code. Table 7 shows the frequencies of replication identified for each unique clone identified by each tool.

Frequency	CCFinder	CloneDr	Covet	JPlag	Moss
1	569	66	40	95	104
2	98	6	34	10	8
3	33	2	13	4	0
4	14	0	6	1	0
5	16	0	5	0	0
6	19	0	5	0	0
7	2	0	1	0	0
8	0	0	1	0	0
9	0	0	0	0	0
10	0	0	0	0	0
11	0	0	0	0	0
12	0	0	2	0	0
13	0	0	1	0	0

Table 7: Total occurrences of each clone per tool

Table 7 shows that it is Covet that recognizes the largest number of replications within the application. The table shows that Covet identified 3 cases where a clone has

been replicated 12 or more times within the source code, the other tools recognize no more than 7 instances.

	CCFinder	CloneDr	Covet	JPlag	Moss
Recall	72	9	19	12	10
Precision	72	100	63	82	73

Table 8: Precision and recall

The results up to now have been based on all the clones identified by the tools. However, not all automatically identified clones are likely to be actual clones. Thus, a subjective assessment was then made of the clones that were identified by the tools which could be considered actual clones for the purpose of preventative maintenance. From a total of 1463, from the initial removal criteria 563 clones were rejected as being false matches clones. Thus, based on this analysis the values of recall and precision could then be calculated. Recall is the percentage of true clones actually identified by each tool. This is shown within Table 8. CCFinder has by far the greatest recall identifying a total of 72% of the true clone base.

A measure of precision has also been made. The table shows that Covet identified the greatest percentage of false clones and therefore had the worst precision with 37% of the clones being identified not actually being clones. Conversely, CloneDr had the best precision with none of the clones later being identified as false matches.

5. Evaluation

The aim of this paper has been to identify which of the evaluated tools are best suited to support the process of software maintenance. In order to address this issues it needs to be considered what requirements are required of such a support tool. The requirements such a tool to support maintenance are considered to be the:

output of a high proportion or all of the clones present within the code
output of a low proportion or no incorrectly identified clones
matching and output of clones that have high frequencies of replication
output of clones that are large in terms of lines of code
output of clones that can be modified or removed with minimum impact to the application
ease of usability of the tool

These points are now considered in relation to the results obtained.

5.1 Output of a high proportion of the clones

The identification of a high proportion of the clones is important for maintenance so that the severity of the modification problem can be addressed prior to maintenance and that proper consideration can be given to the selection and attributing of a priority for the removal of the clones. Within this paper this requirement have been investigated in a number of ways. CCFinder identified more clones than the other tools but the greater proportion of these clones identified was across files. Proportionally CloneDr identified more clones that were internally replicated within a file. However, the most predictive assessment of this requirement is the metric of recall being to percentage of the clones identified from the total known set. CCFinder identified the greatest total number of clones, thus resulting in the highest level of recall 72%.

Overall each tool identified some clones that were not identified by any other tool and that each tool overlapped those that it identified with other tools. In all instances these overlaps were different. Only through using all the tools would it have been possible to identify the total set of clones.

Output of a low proportion of incorrectly identified clones

The output of a low proportion of incorrectly identified clones is important to ensure that the maintenance process is efficient. In most instances false positives will have to be verified manually. This provides a cost in terms of the maintainer's time. For this reason good precision is required of the tools. CloneDr was the only tool who provided perfect precision, thereby identifying no false positive matches, and therefore not resulting in the incurring of wasted maintenance effort. This is due to the automation process for clone removal; if it can't be automatically removed then its not identified as a clone. Thus in this instance a tradeoff has been applied to forsake high recall for perfect precision.

All other tools outputs were found contain at least 1 in 5 clones to be false positives. Of course the greater the number of clones that each tool identifies so the total number of false positives rises and thereby the potential for wasted effort.

In some uses of clone detection for maintenance the identification of false-positives will not be an issue. For instance, consider a scenario when a change is required to a specific portion of code, a search for clones could then be made that match, and only match, the

specifically identified portion of the code to be changed. Since modification are only considered for the matched code it is unlikely that false positives will be identified in this instance.

5.3 Matching and output of clones with a high frequencies of replication

Clones represent the potential for wasted maintenance effort. One way in which preventative maintenance can assist is through the removal of these clones. Therefore, the clones that are replicated most frequently within the code potentially offer the greatest payback for conducting preventative maintenance. Tools that are able to match the largest sets of duplicate functionality potentially offer the greatest payback to maintainers. The results of the analysis process showed that Covet followed by CCFinder best satisfied this requirement. However, the benefits CloneDr's ability to automatically conduct an automated clone replacement process should be not underestimated.

5.4 Output of clones that are large in terms of lines of code

As for the same reasons as indicated above the more code that can potentially be removed per change the greater the potential payback for maintenance. Thus the size of the clones identified is important. The largest clone identified was by Covet at 123 LOC, but the tools generating the largest mean for all clones was JPlag. Overall, however, all tools showed fairly similar performance levels.

Output of clones that can be modified or removed with minimum impact

Impact of change is effectively the maintenance cost for removing each clone. Due to the costs involved in conducting this analysis this requirement was not possible to assess except where the process is known to have been automated as in CloneDr. However, what was possible to assess was the change impact to an application. Where removal of the clone was focused within a source file the program comprehension costs are likely to be less when more files are involved. As indicated above CloneDr identified a very high percentage of clones that were internally replicated within a file.

5.6 Ease of usability of the tool

When running an analysis process other factors need to be taken into account such as ease of use, speed and language support. No subjective measures of the usability of these tools have yet been made, but an indication of factors such as language support was included in table 1.

6. Conclusions and Further Work

The results have identified that there is no single and outright winner for clone detection for preventive maintenance. Each tool had some factors that may ultimately prove useful to the maintainer. Furthermore the ultimate choice is most likely to differ under the circumstances at which the change proposal is made. For instances, whether precision or recall is the most highly desirable requirement or any combination thereof.

What this analysis process has identified is need to be able to accurately define requirements for the identification and removal process and this paper has thus identified a set of criteria upon which this assessment can be based. Furthermore, it has also identified areas of strengths and weaknesses in each tool that may ultimately lead to their improvement. However, due to the plagiarism tools only considering across file duplication these are of less use than the dedicated clone detection tools.

One way in which a more definitive analysis could be performed may be, based on the clone set identified, to investigate a priority for those clones which it would be most beneficial to remove. From this analysis it may be possible to make a more definitive selection of clone identification tool.

A further way in which this process could be improved would be to automate the collation process and to be able to pool the results of using each tool. This work has already been started though this project but the removal of false positives still needs to be addressed.

One way of more effectively dealing with false positives is to improve the process by which they can be identified. Currently the output of most clone detection tools is a simple textual representation. Applying a graphical representation will allow the user to browse summaries of each source code file detailing the clones detected across and within file structures. Plotting such graphical representation will allow maintenance's to more efficiently evaluate and plan the preventative maintenance process of clone removal.

References

[Aik02] Aiken, A., *A System for Detecting Software Plagiarism (Moss Homepage)*, Last visited 11th April 2002

[Bak95] Baker B. S., *On Finding Duplication and Near-Duplication in Large Software Systems*, 2nd Working Conference on Reverse Engineering 1995

[Bax98] Baxter I.D., Yahin A., Moura L., Sant'Anna M., Bier L., *Clone Detection Using Abstract Syntax Trees*, International Conference on Software Maintenance 1998

[Bur97] Burd E.L., Munro M., *Investigating the Maintenance Implications of the Replication of Code*, International Conference on Software Maintenance 1997

[Duc99] Ducasse S., Rieger M., Demeyer S., *A Language Independent Approach for Detecting Duplicated Code*, International Conference on Software Maintenance 1999

[Joh94] Johnson J. H., *Substring Matching For Clone Detection and Change Tracking*, International Conference on Software Maintenance 1994.

[Kam01] Kamiya T., Ohata F., Kondou K., Kusumoto S., Inoue K., Maintenance Support Tools for Java Programs: CCFinder and JAAT, International Conference on Software Engineering 2001

[Kom01] Komondoor R., Horwitz S., *Using Slicing to Identify Duplication in Source Code*, Symposium on Static Analysis 2001

[Lag97] Lague B., Proulx D., Mayrand J., Merlo E., Hudepohl J., *Assessing the Benefits of Incorporating Function Clone Detection in a Development Process*, International Conference on Software Maintenance 1997

[May96] Mayrand J., Leblanc C., Merlo E., *Automatic Detection of Function Clones in a Software System Using Metrics*, International Conference on Software Maintenance 1996

[Nie97] Niessink F., van Vliet H., Predicting Maintenance Effort with Function Points, International Conference of Software Maintenance 1997

[Pre00] Prechelt L., Malpohl G., Philippsen M., JPlag: Finding plagiarisms among a set of programs, Technical Report 2000-1

[Tak96] Takang A., Grubb P., Software Maintenance : Concepts and Practice, Thomson Computer Press 1996, ISBN 1-85032-192-2