# Iteration Method for Clone Detection Using Abstract Syntax Suffix Trees

E.Kodhai
Department of Information Technology
Sri Manakula Vinayagar Engineering College
Puducherry
kodhaiej@yahoo.co.in

Dr.S.Kanmani
Department of Information Technology
Pondicherry Engineering College
Puducherry
kanmani@pec.edu

## ABSTRACT

Copying a code fragment and reusing it by pasting with or without minor modifications is a common practice in software development environments. As a result software systems often have sections of code that are similar, called software clones or code clones. Various techniques have been proposed to find duplicated redundant code.

Most methods for detecting clones are limited to a single revision of a program. Current techniques based on abstract syntax suffix trees find syntactic clones in linear time and space. The incremental detection technique uses token-based clone detection and requires less time to detect clones in each revision separately. But it can only detect the similar clones (type1).

This paper is a proposal for an iteration method for clone detection using abstract syntax trees, which detects all types of clones (1, 2, 3).

## Keywords

Software Clones, Suffix Trees, Clone Detection, Abstract Syntax Trees, Reuse.

## 1. INTRODUCTION

Clones are identical or similar fragments in a software system. Clones are usually created by copy-and-paste programming and extending existing code. Clones of this nature may be compared on the basis of the program text that has been copied. We can distinguish the following types of clones:

• **Type 1** is an exact copy without modifications (except for whitespace and comments).

• **Type 2** is a syntactically identical copy; only variable, type, or function identifiers have been changed.
• **Type 3** is a copy with further modifications; statements have been changed, added, or removed.

Several techniques have been proposed to find these types of clones.

Clones seem to have identical or similar logic, thus, require additional effort from engineers in development. For example, they have to make sure that multiple clones are modified in a consistent manner.

Previous research shows that a significant amount of code (between 7% to 23%) of a software system is cloned code [3, 4, 5, and 6]. While programmers often practice cloning with clear intent [7] and it is beneficial in certain situations [8], one of the major difficulties with such duplicated fragments is that if a bug is detected in a code fragment, all the fragments similar to it should be investigated to check for same bug [9]. Moreover, when enhancing or adapting a piece of code, duplicated fragments can multiply the work to be done [10].

From a program comprehension point of view, clones carry important domain knowledge and thus studying the clones in a system can assist in understanding it [10]. Moreover, by refactoring the clones detected, one can potentially improve understandability, maintainability and extensibility, and reduce the complexity of the system [11].

Fortunately, several (semi-automated techniques for detecting code clones have been proposed. Several studies show that lightweight text-based techniques can find clones with high accuracy and confidence, but detected clones often do not correspond to appropriate syntactic units [12, 13]. Parser-based syntactic (AST-based) techniques, on the other hand, find syntactically meaningful clones but tend to be more heavyweight, requiring a full parser and subtree comparison method. On the other hand, an Incremental detection technique detects clones in less time in each revision separately [2]. Moreover, it only detects the similar clones of type 1.

In this paper, we propose an iteration method for clone detection using abstract syntax trees. The motivation is based on the assumption that only a comparatively small amount of files change per revision, causing a lot of work to be done redundantly when rerunning every part of clones. Therefore, internal results are not discarded, but reused and modified according to the files that changed for the respective revision.

## 2. BACKGROUND

The analysis based token-suffix trees offers several advantages over other techniques. It scales very well because of its linear complexity in both time and space, which makes it very attractive for large systems. Moreover, no parsing is necessary and, hence, the code may be even incomplete and syntactically incorrect. Another advantage for a tool builder is that a token-based clone detector can be adjusted to a new language in very short time [14].

As opposed to text-based techniques, this token-based analysis is independent of layout. Also, token-based analysis may be more reliable than metrics because the latter are often very coarse-grained abstractions of a piece of code; furthermore, the level of granularity of metrics is typically whole functions rather than individual statements.

Two independent quantitative studies by Bellon/Koschke [15] and Bailey/Burd [16] have shown that token-based techniques have a high recall but suffer from many false positives, whereas Baxter's technique has a higher precision at the cost of a lower recall. In both studies, a human analyst judged the clone candidates produced by various techniques. One of the criteria of the analysts was that the clone candidate should be something that is relatively complete, which is not true for token-based candidates as they often do not form syntactic units.

Syntactic clones can be found to some extent by token based techniques if the candidate sequences are split in a post processing step into ranges where opening and their corresponding closing tokens are completely contained in a sequence.

The AST-based technique, on the other hand, yields syntactic clones. And it was Baxter's AST-based technique with the highest precision in the cited experiment. Moreover, the AST-based clone detection offers many additional advantages as already mentioned in the introduction.

Unfortunately, Baxter's technique did not match up with the speed of token-based analysis. Even though partitioning the subtrees in the first stage helps a lot, the comparison of subtrees in the same partition is still pairwise and hence requires quadratic time. Moreover, the AST nodes are visited many times both in the comparison within a partition and across partitions because the same node could occur in a subtree subsumed by a larger clone contained in a different partition.

Another point is that the construction of the AST is more expensive than the generation of a token stream. And with a post processing step the token based approach will lead to similar results in the area of eliminating syntactic incomplete clones. We assume however that the clone detection is part of a larger system (a tool chain, a refactoring tool, or an IDE) and the AST is already available. It would be valuable to have an AST-based technique at the speed of token-based techniques.

## 3. ITERATION METHOD

The First step is to transform the source program into tokens. Since the tokens stored are reused for the next revision, instead using a single token table, multiple token tables are used. i.e for each file a separate token table is created. If a new file is added or deleted it will be easier to be added or deleted. If a file is modified then a new token table is created deleting the old token table.

Before transforming it checks for pervious revision. If available it uses the previous revision details such as token tables, ASTs, suffix tree and detected clones. It just gets the details of the modified files and proceeds further. If not it, records it as the first visit and starts from the scratch.

### 3.1 Constructing AST

The next step is to parse the tokens into ASTs. Then, we serialize the AST by a preorder traversal. For each visited AST node N, we emit N as root and associate the number of arguments (number of AST nodes transitively derived from N) with it (in the following presented as subscript). Note that we assume that we traverse the children of a node from left to right according to their corresponding source locations so that their order corresponds to the textual order.

These ASTs are also stored as an intermediate result. The addition and deletion in AST will be the nodes and edges. For modification it first checks the nodes and edges can be reused or not. If possible it just reuses it instead of deleting and adding the new one. If not it then creates the nodes and edges according to the changes.

### 3.2 Using Suffix trees

Next the AST are converted into generalized suffix trees. The original suffix tree clone detection is based on tokens. In our application of suffix trees, the AST node type plays the role of a token. Because we use the AST node type as distinguishing criterion, the actual value of identifiers and literals (their string representation) does not matter because they are treated as AST node attributes and hence are ignored. The actual value of identifiers and literals becomes relevant in a post processing step where we make the distinction between type-1 and type-2 clones. We do not distinguish type-1 and type-2 clones at this stage.

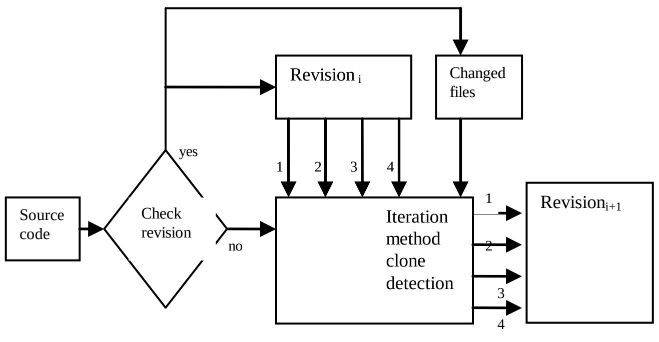
Finally these AST suffix tree are also stored for next revision. In the next revision the changes are made as done in ASTs for the suffix tree.

### 3.3 Detecting Clones

The previous step has produced a set of clone classes of maximally long equivalent AST node sequences. These sequences may or may not be syntactic clones. In the next step these sequences will be decomposed into syntactic clones.

Procedure is used to report clones based on the representative. It may filter clones based on various additional criteria such as length, type of clone, syntactic type (e.g., it may ignore clones in declarative code), differentiates the clone class elements into type-1 and type-2 clones, and finally reports all clones of a class to the user.

These clones are also stored for the next revision. In the next revision, the results of the previous revision are compared to the modified files for the clones.



1. Token tables
2. ASTs
3. Suffix tree
4. Detected clones

**Figure 1. System Architecture**

## 3.4 Approach

The overall task of this proposal is to develop a framework for an iteration method for clone detection using abstract syntax suffix trees that requires less time for clone detection in multiple revisions of a program than the separate application of an existing approach. In addition, a mapping between the clones of every two consecutive revisions must be generated.

The first part of the task addresses the time $t_{all}$ which is needed to analyze n revisions of a program's source code. The assumption is, that time can be saved by eliminating unnecessary calculations resulting from discarding intermediate results. It is desirable to make $t_{all} < n \cdot t_{single}$ true. Instead of starting from the very beginning, the analysis of a revision should reuse and modify results of the previous revision. This requires an overview over all results which are produced during the clone detection process and assessment of whether they might serve for being reused.

Apart from improving the performance, clones of one revision are to be mapped to the clones of the previous revision. In the simplest case, clones remain untouched and no change happens to any clone. On the other hand, clones can be introduced, modified, or vanish due to the modification of the files they are contained in. The different changes that can happen to a clone must be summarized.

When a file is deleted, the token table for the respective file can just be dropped after the suffix tree has been updated. When a file is added, a new token table is created. The location of a token must therefore be extended to a tuple (file, index) instead of just having a single index. File selects the token table for the file in which the token is contained, and index denotes the position of the token inside that file.

The intermediate results from $revision_i$ are the tokens stored in token tables, the suffix tree and the clone pairs. These are given as input together with the files that changed from $revision_i$ to $revision_{i+1}$. Depending on the changes of the source files, the data structures are modified in order to conform to the source code of $revision_{i+1}$. Again, they are kept in memory in order to be reused for the next revision.

## 4. RELATED WORK

Different techniques have been deployed for the detection of simple clones. They can be broadly categorized based on the program representation and the matching technique. For program representation, the different options are plain text [20][21], tokens[23][22], abstract syntax trees[24], program dependence graphs [26][27], and metrics for code structures [25]. The different matching techniques include suffix tree based token matching [23], text fingerprints matching [21], metrics value comparisons [25], abstract syntax trees comparisons [24], dynamic pattern matching [25] and neural networks [28].

A novel approach based on abstract syntax trees was proposed by Baxter *et al.* in [29], which can produce macros bodies to eliminate duplication. Due to its internal representation (ASTs), this tool is strongly language dependent, can be run only against compliable systems and has higher memory requirements than our lightweight approach.

Regarding the concern towards validation in terms of recall and precision of clone detecting tools, Bellon [30] conducted an experiment, whose main concern was to compare the quality of the results provided by several tools ( [29], [32],[33], [31] and [34]). As a conclusion to that experiment, the author stated that there was no absolute winner, every approach implying both advantages and disadvantages.

## 5. CONCLUSION

In contrast to conventional clone detection approaches, Iteration method analyzes multiple revisions of a program. The benefit compared to separate clone detection for each revision is, that intermediate data structures can be reused for the analysis of the next revision. This avoids analyzing parts of the source code again and again which do not change between revisions. Using information about the files that changed from the previous to the current revision, our method modifies the token tables, the generalized syntax suffix tree and the set of clone pairs to conform to the current revision. The result produced is a set of clone pairs for each revision that is analyzed. Moreover, it also detects all the clone type 1,2 & 3, since it is using the syntax suffix trees technique.

## 6. REFERENCES

[1] Nils Gode, Rainer Koschke Incremental Clone Detection – European Conference on Software Maintenance and Reengineering – 2009.
[2] Rainer Koschke, Raimar Falke, Pierre Frenzel, Clone Detection Using Abstract Syntax Suffix Trees– Working Conference on Reverse Engineering – 2006.
[3] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *WCRE*, pp. 86-95, 1995.
[4] I. Baxter, A. Yahin, L. Moura and M. Anna. Clone Detection Using Abstract Syntax Trees. In *ICSM*, pp. 368-377, 1998.
[5] C. Kapser and M. Godfrey. Supporting the Analysis of Clones in Software Systems: A Case Study. *JSME: Research and Practice*,18(2):61-82, 2006
[6] J. Mayrand, C. Leblanc and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics.In *ICSM*, pp. 244-253, 1996.
[7] M. Kim and G. Murphy. An Empirical Study of Code Clone Genealogies.In *FSE*, pp. 187-196, 2005.

[8] C. Kapser and M. Godfrey. “Cloning Considered Harmful” Considered Harmful. In *WCRE*, pp. 19-28, 2006.

[9] Z. Li, S. Lu, S. Myagmar and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE TSE*,32(3):176-192, 2006.

[10] J. Johnson. Visualizing Textual Redundancy in Legacy Source. In *CASCON*, pp. 171-183, 1994.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*.Addison-Wesley, 2000.

[12] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, 33(9):577-591, 2007.

[13] F.V. Rysselberghe and S. Demeyer. Evaluating Clone Detection Techniques. In *ELISA*, 12 pp., 2003.

[14] M. Rieger. Effective Clone Detection Without Language Barriers. Dissertation, University of Bern, Switzerland,2005.

[15] S. Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master’s thesis, University of Stuttgart, Germany, 2002.

[16] J. Bailey and E. Burd. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In SCAM, 2002.

[17] E. McCreight. A space-economical suffix tree construction algorithm. Journal of the ACM, 32(2):262–272, 1976.

[18] E. Ukkonen. On-line construction of suffix trees. Algorithmica,14(3):249–260, 1995

[19] B. S. Baker. Parameterized Pattern Matching: Algorithms and Applications. JCSS, 1996.

[20] Ducasse, S, Rieger, M., and Demeyer, S. A language independent approach for detecting duplicated code. In Proc. Intl. Conference on Software Maintenance (ICSM ’99), pp. 109-118.

[21] Johnson, J. H. Substring Matching for Clone Detection and Change Tracking. In Proc. Intl. Conference on Software Maintenance (ICSM ‘94), pages 120–126.

[22] Baker, B. S. On finding duplication and near-duplication in large software systems. In Proc. 2nd Working Conference on Reverse Engineering. 1995, pages 86-95.

[23] Kamiya, T., Kusumoto, S, and Inoue, K. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Engineering, vol. 28 no. 7, July 2002, pp. 654 – 670.

[24] Baxter, I., Yahin, A., Moura, L., and Anna, M. S. Clone detection using abstract syntax trees. In Proc. Intl. Conference on Software Maintenance (ICSM ’98), pp. 368-377.

[25] Kontogiannis, K.A., De Mori, R., Merlo, E., Galler, M., and Bernstein, M. Pattern Matching for Clone and Concept Detection. J. Automated Software Eng., vol. 3, pp. 770-108, 1996.

[26] Komondoor, R., and Horwitz, S. Using slicing to identify duplication in source code. In Proc. 8th International Symposium on Static Analysis, 2001, pages 40-56.

[27] Krinke, J. Identifying Similar Code with Program Dependence Graphs. In proceedings of the Eight Working Conference on Reverse Engineering, Stuttgart, Germany, October 2001, pp. 301-309.

[28] Davey, N., Barson, P., Field, S., Frank, R., and Tansley, D. The development of a software clone detector. International Journal of Applied Software Technology, 1(3-4): 219-236, 1995.

[29] Ira Baxter, Andrew Yahin, Leonardo Moura,Marcelo Sant’ Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *ProceedingsICSM 1998*, 1998.

[30] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master’s thesis, Universit ̈at Stuttgart, September 2002.

[31] St´ephane Ducasse, Matthias Rieger, and Serge Demeyer.A language independent approach for detecting duplicated code. In Hongji Yang and Lee White,editors, *Proceedings ICSM ’99 (International Conferenceon Software Maintenance)*, pages 109–118.IEEE, September 1999.

[32] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*,28(6):654–670, 2002.

[33] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eigth WorkingConference on Reverse Engineering (WCRE’01)*,pages 301–309. IEEE Computer Society, October 2001.

[34] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo.Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software System Using Metrics*, pages 244–253, 1996.