Volume 2, Issue 7, July 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Detection of code clones using Datasets

Deepak Sethi
Assistant Prof. (CSE)
PIET,Samalkha(Panipat)

Manisha Sehrawat
Student M.Tech(CSE)
PDM, Bahadurgarh

Bharat Bhushan Naib
Assistant Prof. (CSE)
PDM, Bahadurgarh

Abstract: ***Existing researches suggest that the code clone or duplicated code is one of the main factors that degrades the design and the structure of software and lowers the software quality such as readability, changeability and maintainability. Code clones are generally considered harmful in software development, and the previous approach is to try to remove them through refactoring. However, recent research has provided evidence that it may not always be practical, feasible, or cost-effective to eliminate certain clone groups. Copying and pasting source code is common practice, also known as software resuse. Sometimes these clones are modified slightly to adapt them to their new environment or purpose. When programmers copy, paste, and then modify source code, the once-identical code fragments (code clones) can become indistinguishable as the software evolves over time. It is believed that identical or similar code fragments in source code, also known as code clones, have an impact on software maintenance. Whenever a project is developed a number of clone fragments evolve as the development increases and maintenance issues will also increases. Here initially, we develop a project using ASP.net and C# and then we se a clone detection method, called Solid SDD that can detect clones in XML formats and then display the results diagrammatically.***

Keyword: clone detection, dataset, code cloning,

I. Introduction

Software clones appear in code due to reasons like:

- Code reuse by copying pre-existing codes
- Coding styles is similar
- Instantiations of definitional computations
- Failure to use/identify abstract data types
- Performance enhancement of a project
- Accidently using same technique.

Reusing code fragments by copying and pasting with or without minor adaptation is a common activity in software development. As a result software systems often contain sections of code that are very similar, called code clones. Previous research shows that a significant fraction(between 7% and 23%) of the code in a typical software system has been cloned [1, 2]. While such cloning is often intentional [8] and can be useful in many ways [3, 4], it can be also be harmful in software maintenance and evolution [5,10]. For example, if a bug is detected in a code fragment, all fragments similar to it should be checked for the same bug [6]. Duplicated fragments can also significantly increase the work to be done when enhancing or adapting code [7,9]. Many other software engineering tasks, such as program understanding (clones may carry domain knowledge), code quality analysis (fewer clones may mean better quality code), aspect mining(clones may indicate the presence of an aspect), plagiarism detection, copyright infringement investigation, software evolution analysis, code compaction (for example, in mobile devices), virus detection, and bug detection may require the extraction of syntactically or semantically similar code fragments, making clone detection an important and valuable part of software analysis [11]. Over the last decade many techniques and tools for software clone detection have been proposed. In this paper, we provide a qualitative comparison and evaluation of the current state-of-the-art in clone detection techniques and tools, and organize the large amount of information into a coherent conceptual framework. We begin with background concepts,a generic clone detection process and an overall taxonomy of current techniques and tools. We then classify, compare and evaluate the techniques and tools in two different dimensions. First, we classify and compare approaches based on a number of facets, each of which has a set of (possibly overlapping) attributes. Second, we qualitatively evaluate the classified techniques and tools with respect to a taxonomy of editing scenarios designed to model the creation of Type-1, Type-2, Type-3 and Type-4 clones. Finally, we 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. The primary contributions of this paper

are: (1) a schema for classifying clone detection techniques and tools and a classification of current clone detectors based on this schema, and (2) a taxonomy of editing scenarios that produce different clone types and a qualitative evaluation of current clone detectors based on this taxonomy.

A. Clone Detection Process

A clone detector tool must try to find pieces of code of high similarity in a system's source code or text. The main problem is that, it is not known initially which code fragments may be repeated. Thus the detector really should compare every possible code of fragment with every other possible fragment. Such a comparison is expensive from a computational point of view and thus, several measures are used to reduce the domain of comparison before performing the actual comparisons. Even after identifying potentially cloned fragments, further analysis and tool support may be required to identify the actual clones. In this section, an overall summary of the basic steps in a clone detection process is provided. This generic overall picture allows us to compare and evaluate clone detection tools with respect to their underlying mechanisms for the individual steps and their level of support for these steps. Figure 1 shows the set of steps that a typical clone detector may follow in general (although not necessarily). The generic process shown is a generalization unifying the steps of existing techniques, and thus not all techniques include all the steps.

1) Preprocessing: At the beginning of any clone detection approach, the source code is divided and the domain of the comparison is determined. There are three main objectives in this phase:

Remove uninteresting parts: All the source code uninteresting to the comparison phase is filtered out in this phase. For example, partitioning is applied to embedded code to separate different languages (e.g., SQL embedded in Java code, or Assembler in C code). This is especially important if the tool is not language independent. Similarly, generated code (e.g., LEX- and YACC-generated code) and sections of source code that are likely to produce many false positives (such as table initialization) can be removed from the source code before proceeding to the next phase [17].

Determine source units: After removing the uninteresting code, the remaining source code is partitioned into a set of disjoint fragments called source units. These units are the largest source fragments that may be involved in direct clone relations with each other. Source units can be at any level of granularity, for example, files, classes, functions/methods, begin-end blocks, statements, or sequences of source lines.

Determine comparison units / granularity: Source units may need to be further partitioned into smaller units depending on the comparison technique used by the tool. For example, source units may be subdivided into lines or even tokens for comparison. Comparison units can also be derived from the syntactic structure of the source unit. For example, an if-statement can be further partitioned into conditional expression, then and else blocks. The order of comparison units within their corresponding source unit may or may not be important, depending on the comparison technique. Source units may themselves be used as comparison units. For example, in a metrics based tool, metrics values can be computed from source units of any granularity and therefore, subdivision of source units is not required in such approaches.

2). Transformation: Once the units of comparison are determined, if the comparison technique is other than textual, the source code of the comparison units is transformed to an appropriate intermediate representation for comparison. This transformation of the source code into an intermediate representation is often called extraction in the reverse engineering community. Some tools support additional normalizing transformations following extraction in order to detect superficially different clones. These normalizations can vary from very simple normalizations, such as removal of whitespace and comments [18], to complex normalizations, involving source code transformations [19]. Such normalizations may be done either before or after extraction of the intermediate representation.

(i) Extraction

Extraction transforms source code to the form suitable as input to the actual comparison algorithm. Depending on the tool, it typically involves one or more of the following steps.

Tokenization: In case of token-based approaches, each line of the source is divided into tokens according to the lexical rules of the programming language of interest. The tokens of lines or files then form the token sequences to be compared. All whitespace (including line breaks and tabs) and comments between tokens are removed from the token sequences. CCFinder [20] and Dup [21] are the leading tools that use this kind of tokenization on the source code.

Parsing: In case of syntactic approaches, the entire source code base is parsed to build a parse tree or (possibly annotated) abstract syntax tree (AST). The source units to be compared are then represented as subtrees of the parse tree or the AST, and comparison algorithms look for similar subtrees to mark as clones[22].

Control and Data Flow Analysis: Semantics-aware approaches generate program dependence graphs (PDGs) from the source code. The nodes of a PDG represent the statements and conditions of a program, while edges represent control and data dependencies. Source units to be compared are represented as subgraphs of these PDGs.

(ii) Normalization

Normalization is an optional step intended to eliminate superficial differences such as differences
in whitespace, commenting, formatting or identifier names.

Removal of whitespace: Almost all approaches disregard whitespace, although line-based approaches retain line breaks.

Some metrics-based approaches however use formatting and layout as part of their comparison.

Removal of comments: Most approaches remove and ignore comments in the actual comparison.

Normalizing identifiers: Most approaches apply an identifier normalization before comparison in order to identify parametric Type-2 clones. In general, all identifiers in the source code are replaced by the same single identifier in such normalizations. However, Baker [18] uses an order-sensitive indexing scheme to normalize for detection of consistently renamed Type-2 clones.

Pretty-printing of source code: Pretty printing is a simple way of reorganizing the source code to a standard form that removes differences in layout and spacing. Pretty printing is normally used in text-based clone detection approaches to find clones that differ only in spacing and layout.

Structural transformations: Other transformations may be applied that actually change the structure of the code, so that minor variations of the same syntactic form may be treated as similar [19].

(iii) Match Detection: The transformed code is then fed into a comparison algorithm where transformed comparison units are compared to each other to find matches. Often adjacent similar comparison units are joined to form larger units. For techniques/tools of fixed granularity (those with a predetermined clone unit, such as a function or block), all the comparison units that belong to the target granularity clone unit are aggregated. For free granularity techniques/tools (those with no predetermined target clone unit) aggregation is continued as long as the similarity of the aggregated sequence of comparison units is above a given threshold, yielding the longest possible similar sequences. The output of match detection is a list of matches in the transformed code which is represented or aggregated to form a set of candidate clone pairs. Each clone pair is normally represented as the source coordinates of each of the matched fragments in the transformed code.

(iv) Formatting: In this phase, the clone pair list for the transformed code obtained by the comparison algorithm is converted to a corresponding clone pair list for the original code base. Source coordinates of each clone pair obtained in the comparison phase are mapped to their positions in the original source files.

(v) Post-processing / Filtering: In this phase, clones are ranked or filtered using manual analysis or automated heuristics.

Manual Analysis: After extracting the original source code, clones are subjected to a manual analysis where false positive clones are filtered out by a human expert. Visualization of the cloned source code in a suitable format (e.g., as an HTML web page [19]) can help speed up this manual filtering step.

Automated Heuristics: Often heuristics can be defined based on length, diversity, frequency, or other characteristics of clones in order to rank or filter out clone candidates automatically.

(vi) Aggregation While some tools directly identify clone classes, most return only clone pairs as the result. In order to reduce the amount of data, perform subsequent analyses or gather overview statistics, clones may be aggregated into clone classes.

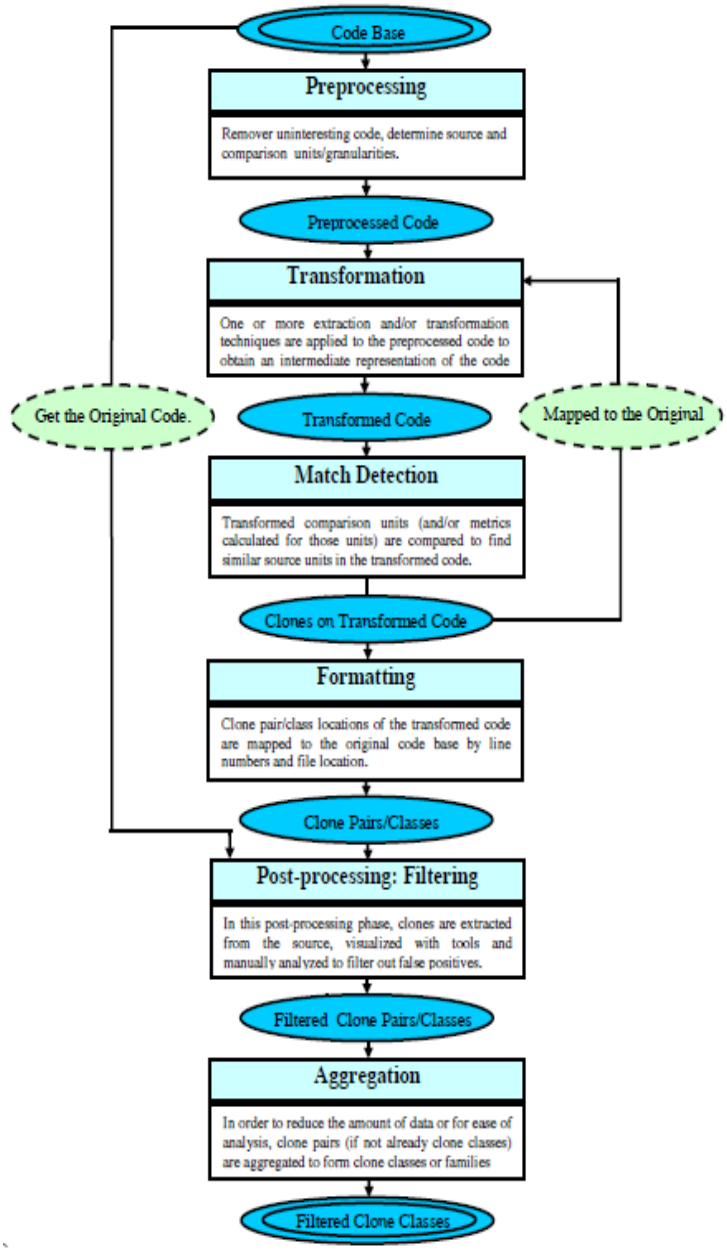

Fig. 1.1 A generic clone detection process

II. Dataset

Dataset is in XML format. Initially, convert the project into dataset .A dataset stores information about **nodes**, hierarchies on these nodes, and **edges** between those nodes. The terms "nodes" and "edges" are used rather than "elements" and "relations", each node and edge has a set of attributes, which are key-value pairs. The key indicates the name of the attribute and is always a string, whereas the value of the attribute can be a string, integer, or float. The attributes encode information about the nodes and edges. For example, a dataset might define a node attribute "type" encoding the type of a source code element, taking on values such as "Class", "Method", and "Field". Another example is an attribute "complexity" that encodes the cyclomatric complexity of a node representing a method. Naturally, not every node or edge need to define the same set of attributes: defining the "complexity" attribute on Field nodes does not make sense for example. Each node in a hierarchy has a parent node and has zero or more child nodes. For example, the children of a Class node are its Methods and Fields, and the parent of a Class is a Namespace node. A dataset might define multiple hierarchies.

III. Methodology

Step 1: Build a project in languages like C#, Java or C++ etc.
Step 2: Load this project in clone detector tool (SolidSDD).
Step 3: Set the parameters like Project Name, Source Folder, Output Folder, language, clone size, gap decay etc.
Step 4: Convert the project into Datasets.
Step 5: Find clones.
Step 6: Create the file view and clone view.
Step 7: Generate report on cloning files and generate clone overview diagrammatically.

IV. Result

First we developed a web application project using ASP.Net, C# and MS Access. Then load the project in Solid SDD tool.

Table I Potential Gain of Code Duplication Removal

Potential gain of duplication removal
657
30.72

Table II Potential Gain of first 5% Duplication Removal

Potential gain of removing the first 5% duplication instances
37
5.63
1.73

Table III Clone Percentage Per Statement

Name	Level	Average Clone %	Average Fan Out	Total #clones	Average #clones	Total #statements	Average #statements
	A	69.95	5	62	1.88	1415	42.88

Figure 4.1 Clone View of a project

Figure 4.1 shows the cloning percentage in terms of colours. Red Nodes are showing cloning(duplicacy) is maximum in these files atleast 42%..Orange nodes shows less duplicacy around 34%. Green nodes shows cloning is less than 17% and gray nodes show there is no duplicacy or cloning. Red colour bi-directional edges shows cloning is maximum in between these two nodes.

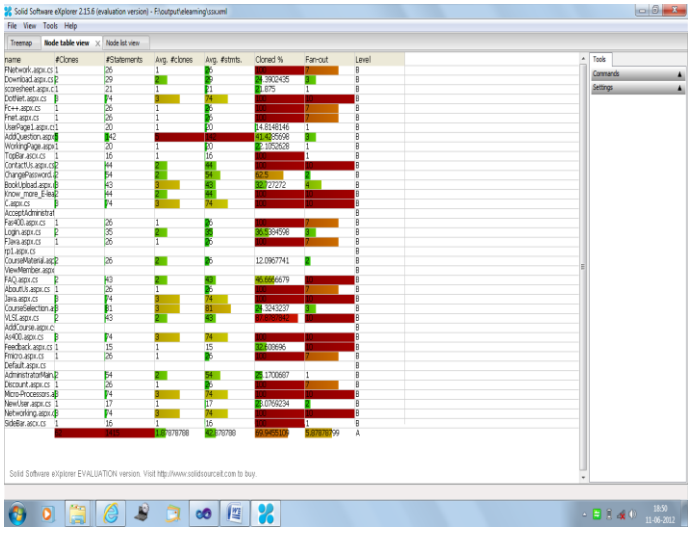

Figure 4.2 Table View of a Project

V. Conclusion

The method can be implemented using standard parsing technology, detects clones in arbitrary language constructs, and detects the number clones without affecting the operation of the program. In this research work the method is applied to a real application of moderate scale, and confirmed previous estimates of clone density of 7-15%, suggesting there is a "manual" software engineering process "redundancy" constant. Automated methods can detect and remove such clones, lowering the value of this constant, at concomitant savings in software engineering or maintenance costs. Clone detection tools also have good potential for aiding domain analysis. We expect to report on more advanced clone detection and removal methods in the near future.this method of clone detection can also be implemented to more complex applications such as web based applications. Solid SDD tool provides a way of visualizing clone detection results in a manner that is observably different from the popular visualization using scatter plots. The solid SDD compliments the standard Text listing and scatter plot visualization and can help users to identify certain characteristics of the clones by graphically isolating clone groups across a list of source files. With the addition of the radial view, the Solid SDD tool has been enhanced to provide a graphical interface.

References

[1] B. Baker, On Finding Duplication and Near-Duplication in Large Software Systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, pp. 86-95 (1995).

[2] C.K. Roy and J.R. Cordy, An Empirical Study of Function Clones in Open Source Software Systems, in: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008, pp. 81-90 (2008).

[3] L. Aversano, L. Cerulo, and M. Di Penta, How Clones are Maintained: An Empirical Study, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, CSMR 2007, pp. 81-90 (2007).

[4] Cory Kapser and MichaelW. Godfrey, "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software, Empirical Software Engineering, Vol. 13(6):645-692 (2008).

[5] H. Li and S. Thompson, Clone Detection and Removal for Erlang/OTP within a Refactoring Environment, in: ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation, in: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, pp. 169–178 (2009).

[6] C.K. Roy and J.R. Cordy, Near-Miss Function Clones in Open Source Software: An Empirical Study, Special issue on WCRE'08, Journal of Software Maintenance and Evolution: Research and Practice, 23 pp., (2009) (submitted).

[7] C.K. Roy and J.R. Cordy, A Mutation / Injection-based Automatic Framework for Evaluating Clone Detection Tools, in: Proceedings of the 4th International Workshop on Mutation Analysis, Mutation 2009, 10pp. (2009) (submitted).

[8] M. Kim, G.Murphy, An Empirical Study of Code Clone Genealogies, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/SIGSOFT FSE 2005, pp. 187-196 (2005).

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

[10] J. Mayrand, C. Leblanc and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, in: Proceedings of the 12th International Conference on Software Maintenance, ICSM 1996, pp. 244-253 (1996).

[11] C.K. Roy and J.R. Cordy, A Survey on Software Clone Detection Research, Queen's Technical Report:541, 115 pp. (2007).

[12] Kevin Jalbert, Jeremy S. Bradbury, "Using Clone Detection to Identify Bugs in Concurrent Software", 26th IEEE International Conference on Software Maintenance(2010).

[13] Anna Corazza, Sergio Di Martino, Valerio Maggio, Giuseppe Scanniello," A Tree Kernel Based Approach for Clone Detection" 26th IEEE International Conference on Software Maintenance(2010).

[14] Philipp Schugerl, Juergen Rilling, Philippe Charland," Reasoning about Global Clones Scalable Semantic Clone Detection", IEEE (2011).

[15] James R Cordy, Chanchal K. Roy, "The NiCad Clone Detector" 19th IEEE International Conference on Program Comprehension (2011).

[16] James R Cordy, Chanchal K. Roy, "DebCheck: Efficient Checking for Open Source Code Clones in Software Systems" 19th IEEE International Conference on Program Comprehension (2011).

[17] M. Rieger. "Effective Clone Detection without Language Barriers", Ph.D. Thesis, University of Bern, Switzerland, 2005.

[18] B. Baker, A Program for Identifying Duplicated Code, in: Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, Vol. 24:4957, 24:49-57 (1992).

[19] C.K. Roy and 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, pp. 172-181 (2008).

[20] T. Kamiya, S. Kusumoto and K. Inoue, CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, IEEE Transactions on Software Engineering, 28(7):654-670 (2002).

[21] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, Comparison and Evaluation of Clone Detection Tools, Transactions on Software Engineering, 33(9):577-591 (2007).
[22] I. Baxter, A. Yahin, L. Moura and M. Anna, Clone Detection Using Abstract Syntax Trees, in: Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, pp. 368-377 (1998).

