

An Efficient Technique for Detecting & Analyzing Structural Clones from Software Libraries

Neha Maggu¹, Komal Arora²

¹M.Tech student, CSE Department, SPGOI, Rohtak
nmaggu20@gmail.com

²Asstt. Professor, CSE Department, SPGOI, Rohtak
grewalkomal14@gmail.com

Abstract

Cloning refers to the redundancy of code in the software system. Software developers prefer cloning as they are under tight time constraints and also for the performance reasons. Duplicated code also has negative impacts in software maintenance & evaluation. Copying of code can also indicate copying of bugs in the software. Simple Clones are formed by similar text fragment but there are large program structures which are formed by the configuration of simple clones, called structural clones. Earlier various techniques have been proposed for detecting similar code fragments in the software, which are called simple clones. But we can obtain further gains by elevating the level of code clone analysis. Recurring patterns of simple clones indicates the presence of higher level similarities i.e. structural clones. In this paper, we will propose a technique to detect some useful types of structural clones. This approach will detect structural clones by using data mining technique. We will describe a tool called clone miner that implements this technique.

Keywords: Structural Clones, Clone detection, Analyzing Clones, Clone Identification from Software Libraries

1 Introduction

A “clone” in software is a segment of code that has been created through duplication of another piece of code. e.g. Copy and paste. Copying code fragments and then reuse by pasting with or without minor modifications or adaptations are common activities in software development. This type of reuse approach of existing code is called code cloning and the pasted code fragment (with or without modifications) is called a clone of the original. However, in a post-development phase, it is difficult to say which fragment is original and which one is copied and therefore, fragments of code which are exactly the same as or similar to each other are called code clones, i.e., instances of duplicated or similar code fragments are called code clones or just clones. Previous research shows that a significant fraction (between 7% and 23%) of the code in a typical -software system has been cloned, and in one extreme case it was even 50%.

While such cloning is often intentional and can be useful in many ways, it can also be harmful in software maintenance and evolution. For example, if a bug is detected in a code fragment, all fragments similar to it should be checked for the same bug. Duplicated fragments can also significantly increase the work to be done when enhancing or adapting code. A recent study that worked in the context of industrial systems shows that inconsistent changes/updates to cloned code are frequent and lead to severe unexpected behavior.

Clones are often the result of copy-paste activities. Such activities are very easy and can significantly reduce programming effort and time as they reuse an existing fragment of code rather than rewriting similar code from scratch. This practice is common, especially in device drivers of operating systems where the algorithms are similar. There are several other factors such as performance enhancement and coding style because of which large systems may contain a significant percentage of duplicated code. There is also “accidental cloning”, which is not the result of direct copy and paste activities but by using the same set of APIs to implement similar protocols. Software components are considered similar to documents stored in libraries.

2 Background

In this section, we provide background on code cloning as a problem in large software systems. We give examples of reasons why code cloning occurs, as well as several examples of problems caused by code cloning.

In addition, we will give an overview of clone detection process and also discuss about the taxonomy of clones i.e. various types of clones which are found in the software system. We will give emphasis on the clone identification, analysis & its removal.

2.1 Code Cloning

Code cloning is considered a serious problem in industrial software. It is suspected that 5 to 10% of many large systems is duplicated code], and has been documented to exist at rates of over 50% in a particular COBOL system]. Code cloning occurs for a variety of reasons: the short term cost of forming the proper abstractions may outweigh the cost of duplicating code; this occurs when the developer is aware of the existence of code that already performs functionality similar to, or the same as, the functionality required. Developers may duplicate code because they are under time constraints; these constraints may be imposed by deadlines, or by LOC performance evaluation. Another likely and reasonable circumstance developers duplicate code is they do not fully understand the problem, or the solution, but they are aware of code that can do some or all of the required functionality.

Several problems can develop as a result of code copying. The size of the source code, and ultimately the size of the object code, may become significantly larger as a result of excessive code cloning. Cloning code can lead to unused, or “dead”, code in the system, which can cause problems with code comprehensibility, readability, and maintainability. Duplication of code may also introduce improperly initialized variables, which may lead to unpredictable behavior of a system, especially if a two clone segments share a common variable. Cloning may be an indication of poor design. Code duplication may indicate design problems such as improper or missing inheritance, or insufficient procedural abstraction. Copying code may also result in copying bugs within the code as well. These effects contribute to “software aging”; over time the program becomes hard to change and possibly less reliable and more inefficient.

3 A Taxonomy of Clones

In the following subsections we present a taxonomy of the types of clones we found during this case study using the clone pairs from CCFinder; in the following section we analyze our findings.

The categories of clones are described using the following template: the first paragraph describes the structure of the clone; the second paragraph describes problems caused by this type of clone; the third paragraph describes reasons why these clones may be introduced into the software; and the fourth paragraph describes a possible solution to that form of cloning activity.

3.1 Duplicated blocks within same function

Characterized as repeated blocks of code within the

same function, these blocks are of non-trivial size (such as 5 to 127 lines of code) and each copy expresses the same semantic idea, generally with very few variables changed (often only one). We found that this type of clone occurs often in the Linux file-system subsystem.

The major problem that this can cause is increased code size; in particular it can cause functions to grow long and unreadable. In addition, this type of cloning may lead to unintended diverging evolution of the code blocks if a developer changes one block, and not another. A bad initialization or ‘value changed’ type of error can very easily happen in this type of code, because it is likely variables used by the blocks in each other’s scope.

Situations where this typically occurs are in control structures such as `switch` and `if/else` statements. The cause of this may be that some developers do not anticipate a condition that may require a similar block, so they do not think to make the block a function from the start. Also, making the function that encapsulates the functionality of this clone block may appear to be too much work, because of the number of local variables involved in the code block. Another reason may be time: it is very fast to just copy and paste the block just a few lines down, and the developer “knows” the code works, so it is a quick and dirty solution. Performance may also be an issue, if many local variables are required to be passed, stack creation and destruction may be time consuming.

A solution to this problem, as with many code clones, would be to create a new function or macro to represent the block, and call the function where these clones occur. Parameters to the function would be the few changed variables that occur in the code block. One would expect this change to be simple and straightforward to implement.

3.2 Similar functions, same file

This type of clone occurs when a programmer has two functions performing very similar tasks, with minor variations. These types of clones are often characterized by changing only a few function calls, variable initializations, constants, or other minor things. We consider any functions which both match 60% of their code to be cloned functions.

Consequences of this type of clone are increased code size and fixing bugs may be harder because same error may be spread across several functions, as well as the functions may evolve on separate paths as various maintainers update them.

Developers are likely to do this when the effort required to parameterize the code block and create a more general function appears to be too great when compared to simply copying the code. Also cloning the function may actually make the program conceptually simpler, because the function names can be specific and

meaningful. This type of cloning we do not consider extremely harmful because clones are not physically far apart, but it is recommended that such cloning activity should be documented as it may not be apparent to future maintainers which functions are clones of each other.

Solutions for this can be very simple, or quite complex.

Possible solutions would be to introduce function pointers to the parameter list, adding more parameters for initialization, etc.

3.3 Functions cloned between files within the same directory

This type of clone occurs when the same functionality is required among multiple files. The majority of code duplication that occurs within a directory (excluding duplication with the same file) is related to duplicated functions, more than 80% of clone pairs that occurred within the same directory (but not in the same file) were related to the duplication of functions. It often occurs with no changes at all to the cloned segment of code, or minor changes such as the function name and some variable or function calls. At times, several constants may be changed, global variables accessed and in these cases a solution is harder to find.

Consequences of this type of clone are code size increase, and error finding and changing. The copied code segments are no longer localized in the same file and can easily be identified, but may be scattered across as many as four or five files. At times, this type of code duplication may contribute to source code that is easier to read. Functions will be easier to understand because they will not include extra logic and flows of control which would be required to restructure a function to encompass the more general functionality required of it to eliminate duplicates. This case is less frequent however, and quite often the use of function pointers or some minor conditional operations would create a function which may perform the desired task.

A simple solution to this is to create a common file to use as a library, and migrate the function definitions and prototypes of the cloned functions to this file. This will work best in the case of exact copies, or clones with minor changes.

3.4 Functions cloned across directories

This type of clone may occur when the same functionality is common among several different components in the software. As with functions cloned within a subsystem, it may entail no changes at all to the cloned segment, or minor changes such as the function name and some variable or function calls. We often saw this type of clone for generic kinds of tasks such as parsing options or outputting errors.

Consequences of this type of clone are code size increase, and may increase labor for error fixing. Also, it may be the case that one developer created one component, and is unaware of the clones existing in the rest of the system. In this case, when an error is found, repairs may not even have a chance to be propagated to the rest of the clones.

This type of cloning may occur when a new subsystem is being created, and the design and implementation is based on previous work of another subsystem.

Creating a set of library function may be the easiest solution, but if the function is cloned only between several clones, the effort put into creating a new library, and maintaining it, to be shared by all components may be more work than it is worth.

3.5 Cloned files (possibly with some changes)

This type of clone occurs when a new program arises with requirements that are very similar to those of an existing software system, and the source code is readily available. For example, when a new file system is introduced to the system, it may be possible to copy another's file, and make only minor changes. We saw a very good example of this when we compared `ext2` and `ext3`, in particular `buffer.c` in both systems. This is a very rare occurrence from what we have seen in the file-system subsystem, but in other systems such as this SCSI subsystem this type of cloning activity seems to be much more frequent [10].

Consequences of this type of cloning can be much more severe than function cloning, because the clone has now introduced a large number of lines of code that are common between the two files, and must be changed together, especially when bug fixing. Because it is likely that there will be some alterations to some of the code, it may not be clear where or how to change the cloned file when reflecting changes that have been made to the original code. Also, this is one of the worst-case scenarios for code size increase. In addition, it is possible that side effects (such as inefficient device usage and settings) can occur if the developer does not fully understand the code that he/she has copied. This may lead to inefficiencies in the code and instability. This type of cloning will occur when speed of development may be a factor, or a developer may not completely understand the problem at hand. We have also seen this when drivers are made for related hardware, although not part of this study.

Solutions to this problem may not be as simple as other cloning types. Because the two files are used on different products or include different features, they may need evolve separately from this point on. As well, changes that have been made to the duplicated code may make it difficult to refactor both subsystems completely just to remove the code duplicates. That said, a workable solution may be to try to take the common invariant code

and place it into a common library file which both subsystems could use. This solution may lead to a slightly more complex architecture.

3.6 Blocks across files

This type of cloning is similar to the first one but it occurs in different files within the same directories or across file systems. Often, in the case of cloning blocks across directories, we see that the cloned block is in fact the remains of what appears to be a cloned function. The function is often changed to suit the developers own personal style and also to meet the specific needs of his/her own project. Based on our observations, we would argue that most clones that occur across files start out as whole function clones and then are manipulated to fit the current project goals until what remains are scattered blocks of code which can still be captured as code duplicates.

The main problem with this kind of clone is when the developer wants to modify or change these blocks of code or when they find bugs, it will be very difficult to fix and change these blocks everywhere else, and it is possible that the developer may be completely unaware of the other clones. If any logic on which this block depends changes, then all the blocks may be harmed, and it may be difficult to find all the blocks affected.

The solution for this problem is relative to the size and number of clones that occurs across files. In certain contexts it might be proper to leave the clones as it is, such as in the case of `if` or `case` statement, sometimes making function calls may break the understanding of the logic of the code. In other cases a common library should be made.

3.7 Initialization and finalization clones

This type of clone occurs within the same file or across file systems when initializing data parameters or cleaning up at end of function; we have found that the main portion of the function can perform quite different tasks. This usually occurs when using the same data types or when performing the same tasks such as memory allocation and de-allocation or variable initialization. Finalization clones often encompass exit conditions and logging.

Problems with this type of clone are much less severe than other clone types, and in many cases are unavoidable. Certainly increased code size may be an issue, but other problems related to code duplication do not seem as large of a concern.

Solutions to this sort of problem may be the use of macros or functions, but this seems too complex for something that is of such little issue.

4 Detection Process

4.1 Clone Detection Process

Clone detection is a non-trivial problem. A clone detector must try to find pieces of code of high similarity in a system's source text. The main problem is that it is not known beforehand which code fragments may be repeated. Thus the detector really should compare every possible fragment with every other possible fragment. Such a comparison is prohibitively 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. Nevertheless, over the decade many different clone detection approaches have been proposed because of the importance of clone detection. In this section, we provide an overall summary of the basic steps in a clone detection process. 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. In the following subsections, we provide a short description of each of the phases.

4.1.1 Preprocessing

At the beginning of any clone detection approach, the source code is partitioned 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.

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.

4.1.2 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, to complex normalizations, involving source code transformations. Such normalizations may be done either before or after extraction of the intermediate representation.

a) 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 and Dup are the leading tools that use this kind of tokenization.

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. Metrics-based approaches may also use a parse tree representation to find clones based on metrics for subtrees.

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. The techniques then look for isomorphic subgraphs to find clones. Some metrics-based approaches use PDG subgraphs to calculate data and control flow metrics

b. 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. Davey et al. use the indentation pattern of pretty printed source text as one of the features of their attribute vectors, and Mayrand et al. use layout metrics such as the number of non-blank lines.

Removal of comments: Most approaches remove and ignore comments in the actual comparison. However, Marcus and Maletic explicitly use comments as part of their concept similarity method, and Mayrand et al. use the number of comments as one of their metrics.

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 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. Cordy et al. use an island grammar to

generate a separate pretty-printed text file for each potentially cloned source unit.

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. For instance, Kamiya et al. remove keywords such as static from C declarations.

4.1.3 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.

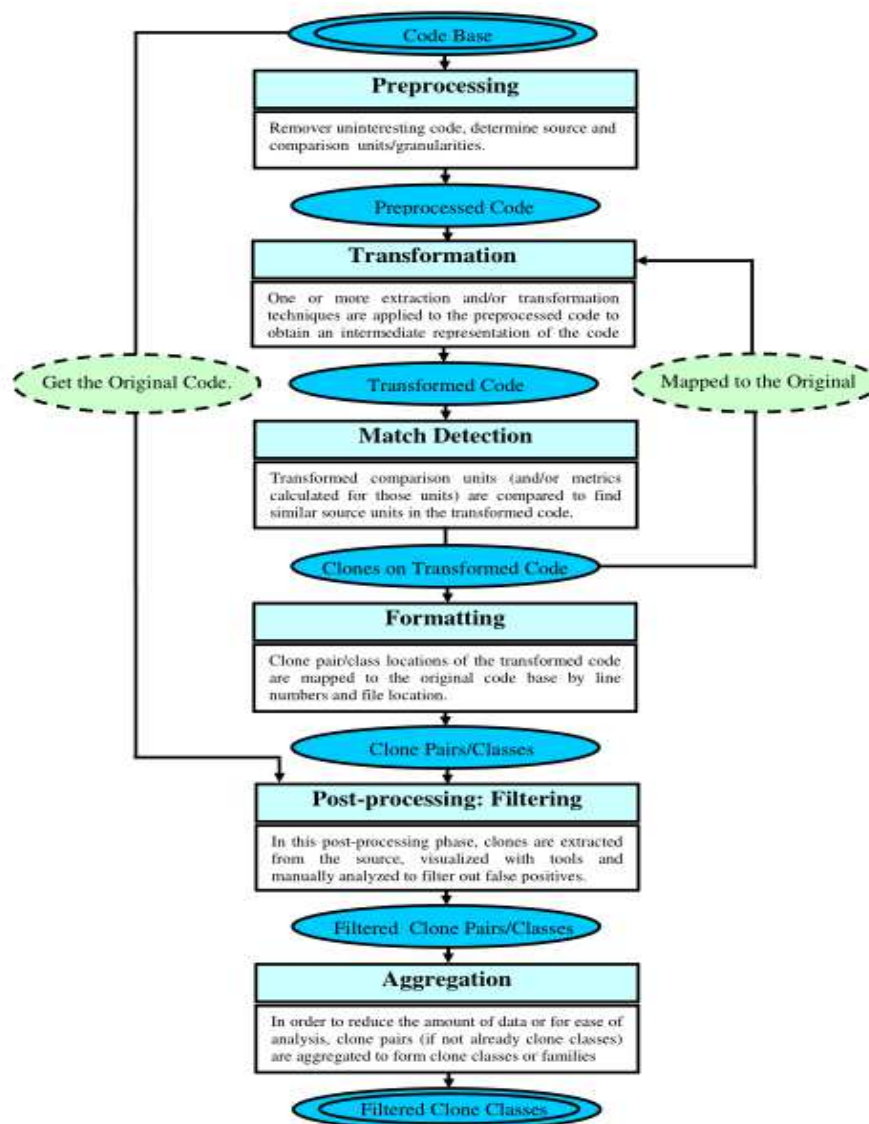


Figure 1: A generic clone detection process

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 represented as the source coordinates of each of the matched fragments in the transformed code.

In addition to simple normalized text comparison, popular matching algorithms used in clone detection include suffix-tree dynamic pattern matching (DPM) [and hash-value comparison

4.1.4 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.

4.1.5 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 or spurious 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 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.

4.1.6 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.

5 Structural Clones

5.1 Introduction

Similar program parts are termed software clones. Simple clones are formed by textually similar code fragments. Larger program structures, formed by configurations of simple clones, are called structural clones. Examples of clones are similar class methods, classes, source files,

directories, any similar software components or recurring patterns (configurations) of similar components.

5.2 Definition of structural clones

A program structure (structure for short) is a connected mixed multigraph where nodes are entities, and (directed or undirected) edges are relationships between entities. A relationship is any meaningful physical or logical connection between two entities in a structure. In a mixed multigraph, the same pair of nodes can be connected by multiple edges which is useful in characterizing certain types of structures.

Definition: A structural clone relation holds between two structures S1 and S2 if (and only if):

- (a) S1 and S2 have the same graph structure
- (b) a clone relation has already been established between corresponding entities in S1 and S2, and
- (c) Corresponding relationships in S1 and S2 are of the same type.

5.3 Benefits of the structural clone concept

For program understanding: As a structural clone comprises many simple clones, structural clones are bigger in size and smaller in number than simple clones, and often embody application domain or design concepts at a higher level than code fragments. Therefore, structural clones provide a simpler and higher-level window into the software similarity situation than simple clones.

For plagiarism detection: Plagiarizers often take “anti-detection” measures such as renaming, reformatting, reordering, scattering (by extracting functions/classes), pruning (removing unused code), reducing generality, inlining, changing comments, and changing string literals (e.g., error messages). The general effect of these anti-detection measures is to break the clone into smaller pieces and scatter them. Structural clone detection and analysis has a higher chance identifying such “disguised” clones, when compared to a simple clone detection/analysis.

For refactoring: Refactoring can remove some clones. Structural clone concept can help in refactoring. Consider two files having a number of cloned methods but occurring in different order in each file. If we focus only on simple clones, these similar methods would appear as candidates for the “extract method” refactoring. However, if we also focus on structural clones, we could also use “extract interface” (or “extract super/abstract class”)

Table 1: Types of Structural Clones found by the Clone Miner

Level 1	Repeating groups of simple clones	
	A	In the same method
	B	Across different methods
Level 2	Repeating groups of simple clones	
	A	In the same file
	B	Across different file
Level 3	Method Clone Sets	
Level 4	Repeating groups of method clones	
	A	In the same file
	B	Across different files
Level 5	File clone sets	
Level 6	Repeating groups of file clones	
	A	In the same directory
	B	Across different directories
Level 7	Directory clone sets	

refactoring here, in addition to the “extract method” refactoring. We can use the “template method” design pattern when we have similar methods that follow the same high-level algorithm but have implementation variations. Typically this translates to a structural clone of code fragments, contained in a method, and having the same order of appearance, but with gaps between them.

For clone detection: De Lucia et al use textual similarity among whole web pages in terms of the Levenshtein distance to decide whether two web pages are clones of each other or not. In contrast, our approach considers the internal structure of the code files, in terms of simple clones shared by the files, to decide about their similarity. If the order of methods in the classes does not matter, then, unlike textual analysis, structural analysis can establish clone relation among classes that differ in the order of methods in class definition only. Without the details of the low-level similarities in the large granularity clones, it is not always straight-forward to take remedial actions like refactoring, as it requires a detailed analysis of low-level similarities. Structural clone concept may also improve the precision and recall of clone detection in

general. Clone detectors usually detect clones larger than a certain threshold (e.g., clones longer than 5 LOC).

Higher thresholds risk false negatives, while lower thresholds detect too many false positives. In comparison, a structural clone detector can afford to have a lower threshold than a simple clone detector, without returning too many false positives

6. Research Methodology

6.1 Structural Clones Detection

This technique is based on the strategy of applying a follow-up analysis to simple clones data. We observed that at the core of the structural clones, often there are simple clones that co-exist and relate to each other in certain ways. This observation is the basis of our proposed work on defining & detecting structural clones. In this technique we will detect some specific types of

structural clones by repeated combinations of co-located simple clones. This structural clone detection technique is implemented in a tool called clone miner implemented in C++. Clone Miner has its own token-based simple clone detector. Our structural clone detection technique works with the information of simple clones obtained from any clone detection tool. This technique only requires the knowledge of simple clone sets (SCSets) and the location of their instances in programs.

In this current work, we will improve the structural clone detection methodology to cover more types of structural clones. The description of structural clone detection techniques is detailed enough to make it possible for others to adopt and further advance elements of our approach..

6.2 Structural Clones Detected by Clone Miner

The concept of structural clones covers all kinds of large granularity repeated program structures. Our Clone Miner can only find some specific types of structural clones, which are listed in TABLE 1. We focussed on these specific types of structural clones because their detection required only lexical analysis, making our method minimally language dependent. Furthermore, such structural clones can be easily detected by well-known data mining techniques. Following Figure shows the hierarchical process of detecting higher-level structural clones given in TABLE 1 from the corresponding lower-level clones. This process begins with the simple clones at the bottom of the figure. Similar to the simple clone sets (SCSets), we have method clone sets (MCSets at level 3), file clone sets (FCSets at level 5) and directory clone sets (DCSets at level 7), which consist of groups of cloned entities at successively higher levels of abstraction. The other types of clones listed in TABLE 1 consist of recurring groups of simple clones, method clones or file clones.

6.3 From Simple Clones to structural Clones

Clone miner firstly find the simple clones in performing structural clone detection and then gradually raising the level of clone analysis to larger similar program structures. The overall algorithm for structural clone detection at various levels is shown in Fig. 2.

6.3.1 Simple clone detection

Simple clone sets (SCSets) are formed by grouping of similar code fragments. The output from certain simple clone detectors is in the form of clone pairs. However, we can easily form SCSets by grouping clone pairs in such a way that every member in a set is a clone of every other member.

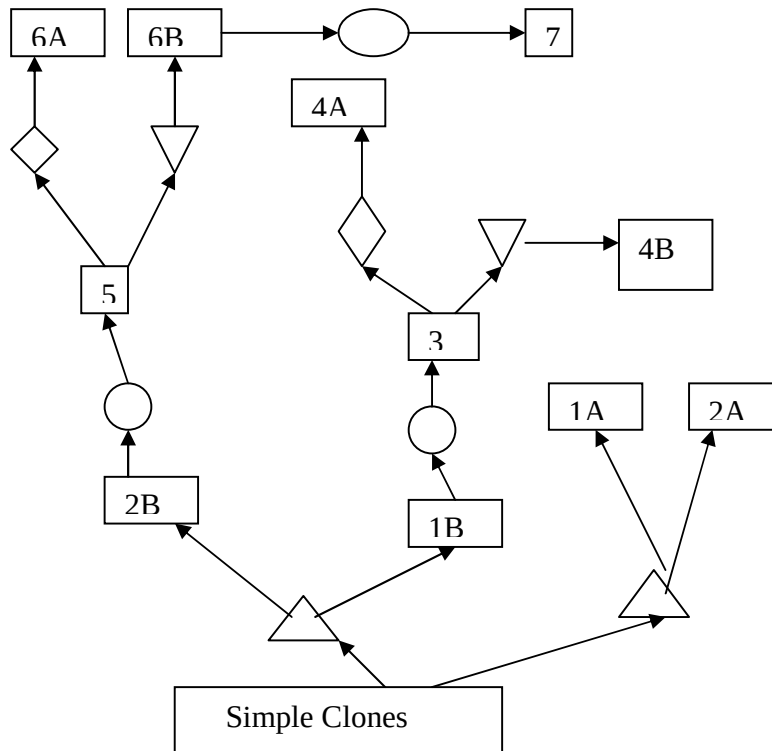


fig2: Hierarchy of structural clones detected by Clone Miner and the overall detection process

For the method-based structural clones (level 1, level 3 and level4 structural clones from TABLE 1), the locations of the methods need to be provided. If the simple clone detector is based on parsing or lexical analysis, this information can be obtained directly, otherwise we can deploy program analysis tools to obtain this information Clone Miner uses Repeated Tokens Finder (RTF), a token-based simple clone detector, as the default front-end tool [6]. RTF tokenizes the input source code into a token string, from which a suffix array based string-matching algorithm directly computes the SCSets, instead of computing them from the clone pairs. RTF currently supports Java, C++, Perl, and VB.net. RTF also performs some simple parsing to detect method and function-boundaries.

6.3.2 Re-organizing the Simple Clone Data

Simple clones data extracted from software systems is manipulated & converted into structural clones at level 1

and 2. Firstly data should be reorganized to make it compatible with the input format for the data-mining technique that is subsequently applied on this data. In this, we will list simple clones for each method or file, depending on the analysis level. For the method level analysis to work, we should know the method or function boundaries in the system and the simple clones are contained within those boundaries, without straddling them. By using this arrangement of simple clones, we get a different view of the simple clones' data, with simple clones arranged in terms of methods or files. A sample of this format is shown in Fig. 3. The first row of data means that the file No. 12 contains three instances of SCSet 9 and one instance each of SCSets 15, 28, 38, and 40. The interpretation is same for the other rows. In this stage, we can easily filter out methods or files that do not participate in cloning at all (i.e. contain no simple clones). These non-representative methods or files are not apparent from the original representation of simple clones' data in terms of clone sets and clone instances.

File ID	SCSet Instances Present
.....	
12	9,9,9,15,28,38,40
13	12, 15, 40, 41, 43, 44, 44
14	9, 9, 9, 12, 15
.....	

Table 2

From this data, we can detect recurring groups of simple clones in different files or methods, to identify the level 2-B and level 1-B structural clones, respectively. The order of different simple clones in which it appear in a file or method is ignored at this stage. If code fragment A1 appears before code fragment B1 in File 1, while code fragment B2 appears before code fragment A2 in File 2, with A1 and A2 being clones of each other and the same for B1 and B2, we may still be interested in finding the clone pattern A1-B1 and A2-B2. The list of SCSets in a file or method is then sorted to facilitate further analysis.

6.3.3 Finding Repeating Groups of Simple Clones

To detect recurring groups of simple clones in different files or methods, we will apply the same data mining technique which is used for “market basket analysis” [22]. The idea behind this analysis is to find the items which are usually purchased together by different customers from a departmental store. The input database consists of a list of transactions, each transaction containing items bought by a customer in that transaction. The output consists of groups of items which are most likely to be bought together. The analogy in this analysis is that a file or a method corresponds to a transaction and the SCSets, represented in that file or method, correspond to the items of that transaction. Our objective here is to find all those groups of SCSets whose instances occur together in different files or methods.

Algorithm for finding files containing the frequent itemset

1. For every frequent itemset frit
 - a. Take the first SCSet
 - b. List all files that contain instances of this SCSet
 - i. For each file
 1. If frit is a subset of the simple clones represented in this file.
 - a. Keep the file in the list, else prune it
 - c. Output the final list

7. Tool Implementation

Clone Miner implements the structural clone detection techniques as discussed. Clone Miner is written in C++, and it has its own token-based simple clone detector. For frequent closed item-sets mining (FCIM), currently we are using the algorithm.

For manipulation of clones’ data, Clone Miner makes use of the STL containers from the standard C++ library. The output from Clone Miner will be generated in the form of text files, so that any visualization tool developed in the future can easily interface with Clone Miner.

For performance evaluation, we ran Clone Miner on full J2SE 1.5 source code³, consisting of 6,558 source files in 370 directories, 625,096 LOC (excluding comments and blank lines), and 70,285 methods, using different values of minimum clone size. For forming FCSet and MCSet, a 50 tokens value is used for the clustering parameter minLen, where the Len is measured in terms of tokens. Likewise, for minCover, a value of 50% is used in all cases. The tests were run on a Pentium IV machine with 3.0 GHz processor and 1 GB RAM.

Each time it takes around two to three minutes to run the whole process from finding simple clones to the analysis of files, methods and directories for structural clones, as mentioned above. The results are given in TABLE 2, in which column headings indicating the minimum size of the simple clones used in each run. All times are calculated in seconds and the memory is calculated in kilobytes.

Table3: PERFORMANCE OF STRUCTURAL CLONE DETECTION

	35 TOKENS	40 TOKENS	50 TOKENS
NO. OF SCSets	8,514	6,156	3,578
CPU TIME TAKEN	137s	134 s	126 s
PEAK MEMORY USED	406,772KB	308,672 KB	235,356 KB

The results from TABLE 2 show that our technique is efficient and scalable. Considerably large systems can be analyzed using reasonable computing resources.

8. Conclusion and Future Work

In this current work, we emphasized the need to study code cloning at a higher level. We introduced the concept of structural clone as a repeating configuration of lower-level clones. We presented a technique for detecting structural clones. The process starts by finding simple clones (that is, similar code fragments). Increasingly higher-level similarities are then found incrementally using data mining technique of finding frequent closed itemsets, and clustering. We implemented the structural clone detection technique in a tool called Clone Miner

While Clone Miner can also detect simple clones, its underlying structural clone detection technique can work with the output from any simple clone detector. We evaluated the performance of Clone Miner and assessed its usefulness by analyzing structural clones found in a number of commercial and public domain software systems. We believe our technique is both scalable and useful. Structural clone information leads to

better program understanding, helps in different maintenance related tasks, and points to potential reusable components across a Product Line. Structural clones are also candidates for unification with generic design solutions. After such unification, programs are easier to understand, modify and reuse.

In the future work, we intend to extend our technique for finding other, more complex types of similarities, and to form a taxonomy of those structural clones. Experimentation with recovery of higher-level design similarities in various application domains, and performing analytical studies to measure the precision and recall of the technique is also part of our future work. Implementing good visualizations for higher-level similarities is currently underway. Analysis of clones can also be much facilitated by querying the database of clones. We have already developed a mechanism of creating a relational database of structural clones' data and a query system to facilitate the user in filtering the desired information. Currently, our detection and analysis of similarity patterns is based only on the physical location of clones. With more knowledge of the semantic associations between clones, we can better perform the system design recovery.

References

- [1] B. Laguë, E.M. Merlo, J. Mayrand, and J. Hudepohl. "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process", Proc. IEEE Int'l Conf. on Software Maintenance (ICSM) '97, Bari, Italy. Oct. 1997.
- [2] B.S. Baker. "A program for identifying duplicated code". In Proceedings of Computing Science and Statistics: 24th Symp. Interface, 1992.
- [3] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software System", Proc. Second IEEE Working Conf. Reverse Eng., July 1995.
- [4] Carter, S., Frank, R.J. and Tansley, D.S.W. (1993), "Clone Detection in Telecommunication Software Systems: A Neural Net Approach", in Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications.
- [5] Cory Kasper and Michael W. Godfrey. "Toward taxonomy of clones in source code: A case study". In Evolution of Large Scale Industrial Software Architectures, 2003.
- [6] Filip Van Rysselberghe and Serge Demeyer. "Evaluating Clone Detection Techniques".
- [7] Filippo Lanubile and Teresa Mallardo. "Finding Function Clones in Web Applications".
- [8] G.A. Di Lucca, M. Di Penta, A.R. Fasolino and P. Granato, "Clone Analysis in the Web Era: an Approach to Identify Cloned Web Pages", Proc. Of the Seventh IEEE Workshop on Empirical Studies of Software Maintenance, Florence, Italy, November 2001.
- [9] De Lucia, A., Francese, R., Scanniello, G., and Tortora, G. Reengineering web applications based on cloned pattern analysis. In Proceedings of the 12th Intl. Workshop on Program Comprehension (IWPC). pages 132-141, 2004.
- [10] De Lucia, A., Scanniello, G., and Tortora, G. Identifying clones in dynamic web sites using similarity thresholds. In Proceedings of the International Conference on Enterprise Information Systems (ICEIS), pages 391-396, 2004.
- [11] Ducasse, S., Rieger, M., and Demeyer, S. A language independent approach for detecting duplicated code. In Proceedings of the International Conference on Software Maintenance (ICSM), pages 109-118, 1999.
- [12] Fowler, M. Analysis Patterns: Reusable Object Models. Addison-Wesley, 1997.
- [13] Fowler, M. Refactoring - Improving the Design of Existing Code. Addison Wesley, 1999.
- [14] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1997.
- [15] Gil, J. Y. and Maman, I. Micro Patterns in Java Code. In Proceedings of the 20th Object Oriented Programming Systems Languages and Applications (OOPSLA), San Diego, CA, USA, pages 97-116, 2005.
- [16] Jens Krinke. Identifying similar code with program dependence graphs. In Proc. Eighth Working Conference on Reverse Engineering, 2001.
- [17] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In Proceedings of ICSM. IEEE, 1998.
- [18] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In Proceedings of the Sixth International Software Metrics Symposium 1999.