Aiding Comprehension of Cloning Through Categorization

Cory Kapser and Michael W. Godfrey
Software Architecture Group (SWAG)
School of Computer Science, University of Waterloo
{cjkapser, migod}@uwaterloo.ca

Abstract

Management of duplicated code in software systems is important in ensuring its graceful evolution. Commonly clone detection tools return large numbers of detected clones with little or no information about them, making clone management impractical and unscalable. We have used a taxonomy of clones to augment current clone detection tools in order to increase the user comprehension of duplication of code within software systems and filter false positives from the clone set. We support our arguments by means of 2 case studies, where we found that as much as 53% of clones can be grouped to form Function clones or Partial Function clones and we were able to filter out as many as 65% of clones as false positives from the reported clone pairs.

1. Introduction

Code duplication, or code cloning, is generally believed to be common in large industrial systems [3, 8, 10, 15, 19, 20, 22]. Management of code cloning and the various problems associated with it is important for the successful evolution of software systems. One such problem is the copying of assumptions that may or may not be documented. When the environment changes to violate these assumptions it may not be clear where in the code changes need to be made. For example, suppose there is an assumption of the existence of a previously initialized global array that has been copied, if the array is now to be initialized within a function bugs will be introduced into the system unless specific measures are taken. The effort to find the required changes may be quite daunting in large industrial systems. To ensure the successful evolution of large software systems, such problems must be addressed. Clone detection methods are a useful mechanism in this situation.

One problem with many clone detection methods is that they return many clones with little or no additional information about them to aid the user in their interpretation. Viewing and classifying thousands of clones manually is time consuming and impractical, but necessary if one hopes to manage clones successfully, by either removing or documenting important points of duplication. The technique we propose attempts to solve this problem by categorizing clones automatically into meaningful groups that encapsulate location and functional attributes of the code involved in the clone.

Another problem with clone detection methods is the trade off between precision and recall.[1] We present an approach that augments and can be augmented to those methods with low precision and high recall in hopes of improving the precision without affecting recall. By grouping clones in regions that they appear in, and classifying them based on the types of regions and the similarity of those regions, we are able to filter many false positives automatically. We also provide users contextual information about the clones, further aiding them by allowing them to manually filter results faster and more precisely.

In our study we have used the tool `CCFinder` [15] to gather the clones from the candidate software system and then we apply our own software tool, called *Clone Interpretation and Classification System* (CICS), to the results. These results are displayed in a GUI that allows the user to browse clones by type, remove clones and clone classes from the analysis, modify which clones should be viewed, and save the current state of the clone analysis session to a database.

There are three major contributions of this work. First, we show two studies that reveal in more detail the characteristics of code duplication in real software systems. This contribution is directed toward research on clone detection. With a better understanding of the characteristics of cloning in real world settings, we can better evaluate and apply clone detection methods in various software engineering settings. Second, we show categorizing clones returned by a detection tool can improve navigation through clones

[1]Precision refers to the percentage of false positives occurring in the results, and recall refers to the fraction of real clones found out of all real clones in the system.

as compared to browsing uncategorised clones. Users can navigate to clones in a task oriented manner. For example, a maintainer bent on removing large sections of duplicated code can quickly locate *Function clones*, *Partial-Match Function clones* and *Cloned functions* rather than browse clones pair by pair. The third major contribution is our approach to filtering. We have shown that filtering clones that occur in specific types of regions is an effective method of removing false positives in the detection results.

The rest of this paper is structured as follows: section 2 outlines the motivation for our research, section 3 describes our method of clone extraction and categorization, section 4 describes the results of our case study, section 5 outlines some general observations we made, section 6 describes some related work, and section 7 summaries our work.

2. Background

In this section, we provide background on code cloning as a problem in large software systems, and how it affects the evolution of such systems. We give examples of reasons why code cloning occurs, as well as several examples of problems caused by code cloning. Then we will discuss strategies when dealing code cloning and the trade offs they present.

We also discuss some of the work leading up to our taxonomy of clones and our current method for clone classification and filtering.

2.1. Code Cloning

Code cloning is considered a serious problem in industrial software [2, 9, 3, 8, 10, 13, 14, 15, 19, 20, 22]. It is suspected that 5 to 10% of many large systems is duplicated code [4, 10], and it has been documented to exist at rates of over 50% in a particular COBOL system [10]. Code cloning occurs for a variety of reasons[3, 8, 14, 15, 20, 22]: 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 where 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. Duplicates can also be introduced with good intentions. Duplicating code can be used to keep software architectures clean and understandable. Duplicates can also be used to keep unreadable, complicated function behavior from entering the system. Creating generic functions or introducing macros to remove clones can create complex code that is hard to read.

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[3, 14]. Cloning code can lead to unused, or "dead", code in the system, that left unchecked can cause problems with code comprehensibility, readability, and maintainability over the life time of the software system[14]. 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 [14]. Code duplication may indicate design problems such as improper or missing inheritance, or insufficient procedural abstraction[8]. Copying code may also result in copying bugs within the code as well. These effects contribute to "software aging" [14]; over time the program becomes hard to change and possibly less reliable and more inefficient, and ultimately the unsuccessful evolution of a software system.

Management of code clones is important and has several avenues. Removal of duplicated code is one solution, but for clones introduced with motives of organization or clarity, this solution is counter productive. Duplicated code does not always require the removal of a duplicate. Documenting duplicates is another solution that can lead to the proper management of the clones. Such documentation would ensure that copies would be considered when modifications were performed on one.

2.2. Comprehending Clones

While clone detection is an area of active research, and several tools exist to facilitate code clone detection, there has been relatively little empirical research on the types of clones that are found, or where they are found. Such research could lead to improvements in detection techniques, the creation of benchmarking suites, and improvements in comprehension tools for displaying clones to the user.

In [16, 17] we began to study clones in several larger software systems, beginning with the Linux kernel file-system subsystem. This study led to several observations. First, clones between structures posed significant problems in regards to false positives. Most clones involving unions, structs, and enumerations were false matches and could be removed. Second, grouping clones by function was an excellent way of ranking closeness of functions and generating function clones. In [17] we found that this method of finding function clones was superior to exact match metric matching used in [2, 9].

These findings led us to believe that by augmenting parameterized token matching clone detection techniques with discriminating filters and by post-processing the results using regional information, we could improve the accuracy and comprehensibility of detected clones. To date we have built a prototype GUI and post-processing suite based on our taxonomy to work with the tool `CCFinder`.

3. Clone Gathering and Classification

Our approach begins with detecting clones in source code. Then regional analysis of the source files is performed and clones are grouped by the region they occur in. Next filters are used to remove clones that are very likely false positives. Then we apply our taxonomy to the groups of clones and return the resulting classification. In the following section we describe the tool `CCFinder` [15] and how it is used in gathering and classifying our clones, and then we will describe in more detail how we perform the classification and filtering of the clones.

3.1. Initial Clone Detection

For our purposes, detection of clones can be done using many different approaches with similar results. In this study we have used the tool `CCFinder`, developed by Kamiya et al. [15], for the initial detection of clones. The tool uses a parameterized matching algorithm to search for code clones within C/C++, Java, and COBOL files. This type of clone detection is good at finding clones with name substitution and line structure changes; the former can cause problems for line by line matching algorithms. Baker introduced a similar algorithm in [3].

The tool `CCFinder` begins by performing a lexical analysis of the source code, resulting in the creation of a list of tokens as part of the syntax of the given programming language. The tokens of all the files are concatenated into a single string. As part of the code transformation, all white space and comments are removed from the string. Next, several language specific transformation rules are applied. Then type, variable, and constant identifiers are replaced by a special identifier (such as `$P`).

Once the source code has been transformed into this abstract token stream, an exact match algorithm is performed to find maximal matching strings within the transformed code. This is done by constructing a suffix tree and locating matching substrings within the tree, as proposed by Baker [3, 4]

After the exact matches have been found, parameter matching is performed. That is, starting from the beginning of a pair of exactly matched transformed strings, `CCFinder` begins parameter matching of the parameters on each line. As the parameters are matched, if a conflict is found but a sufficiently large number of lines have been matched, the clone is reported, and parameter matching begins again after the line creating the conflict.

The results of this process are output into a structured text file that becomes the input for the next step in our process.

3.2. Classifying and Filtering Clones

The following sections will describe our filtering and classification methods. We then describe the taxonomy that we used to classify the clones we saw.

The process that follows is embodied in clone exploration software called the `Clone Interpretation and Classification System` (CICS). This system reads clones from a text file and then processes the input to be used in a graphical interface for clone visualization. Currently it works with the languages C and C++ but is easily extended to support Java. The system is written in python and currently supports the `CCFinder` version 7 output format.

3.2.1. Extracting Regions from Source Code. In our first step we use `ctags` [1], a tool for extracting indices of language objects found in the source code. We then manipulate and augment the data to suit our needs. We find the end points of functions, macros, structs, unions, and enumerations. Then we join consecutive objects of the same type if they are type definitions, prototypes, or variables into regions. These regions have a start point of the first object of one type, *i.e.*, prototypes. Their endpoint is that last consecutive object of the same type.

Using this information we map the contents of the file to eight types of regions: consecutive type definitions, prototypes, and variables; individual macros, structs, unions, enumerations, and functions. Each region is named either by the first object in its list, as is the case for the first four region types, or by the name of the object, as is the case for the final four region types.

In many clone detection tools, comments are not considered as part of the clone detection process. For this reason we do not create regions for them. When creating a region that is a list, such as several consecutive *#defines*, comments are considered to be white space.

3.2.2. Mapping Clone Pairs to Regions. In the next step of the process, for each clone pair we map both segments of the clone to a region in a file. We consider a segment's region to be the one that contains the largest portion of its code. The tool `CCFinder` ends clones that are part of a function at the end of the function, so we are not concerned about clones that may map to several functions. In the case where a clone maps to several different region types, it may be better to break the clone up, but in practice we find the current method to work quite well.

If two regions have cloning between them, we say they have a *cloning relationship*. For each region with a cloning relationship we group together all the clones that form this relationship, and we call this a *Regional Group of Clones* (RGC). An RGC is useful for three reasons. First, filters can be made based on the type of regions they are. For example, we have used filters on structural C constructs such as *structs* and *unions*. When using RGCs for displaying clones, a user can gain more insight into the degree of the relationship between two regions. For example, in our GUI described below, we display not only a selected clone pair, but we also highlight all code which is common between both regions. A third reason is that knowing what type of region the clone came from will aid in user exploration of clones. A user may choose to overlook clones that occur in a *switch-statement* because they know that there tends to be mostly false positives in *switch-statements*.

3.2.3. Filtering RGCs. After we have grouped all of the clone pairs, we classify them by their types and filter them according to certain criteria. Different regions have different structural characteristics, making them more or less prone to false positives in certain clone detection techniques. For example, parameterized clone detection techniques will often report two structures as clones when common sense indicates they are not. This is because of the simple structure of the code in C structs and the nature of parameterized string matching algorithms.

The filters we currently implemented are on structs, union, type definitions, variables, and prototypes. If an RGC has one region of the previously mentioned types, any clone in this RGC must have a minimum of 60% of its lines match exactly. This filter eliminates a substantial number of clone pairs from the result set without removing many, if any at all, true positive matches. We chose 60% after several trials with other values. We found that this threshold works well with long and short clones, and experience indicates that it is low enough to not remove true clones but high enough to filter out most false positives.

3.2.4. Sort Clones into Taxonomy and display results. Once we have a set of initially filtered clones, we can sort them into the taxonomy. After sorting, clones are displayed in a GUI that allows easy navigation of the clones sorted by their type. Clones of the same type in the same region are grouped together for ease of understanding, and all clones that occur in a selected RGC are highlighted to show the degree of cloning in these regions.

3.3. A Taxonomy of Clones

The taxonomy, as described in the following subsections is a hierarchical classification of the clones using attributes such as location and functionality. Its goal is to reflect the types of clones that occur in software systems, and is based on manual inspection of detected clones in several software systems [17].

3.3.1. Partition by Location. The first level of the taxonomy partitions the clones based on the location of the two code segments within the software system. Clones that occur in the same region of the same file are called *Same Region Clones*. Clones that have both code segments within the same file are called in *Same File Clones*, while clones that span different files but in the same directory are called in *Same Directory Clones*. Clones where the code segments are in different directories are placed in *Different Directory Clones*. This partitioning is significant as we have found that the dominant traits of each of these groups are different. For example, clones in the group *Same Directory Clones* tend to be function clones, clones in *Different Directory Clones* tend not to be function clones, and the *Function Clones* that are in different directories appear to be changed more.

3.3.2. Partition by Region. The next level of the taxonomy partitions each of the above groups, with exception of *Same Region Clones*, by the types of regions in the RGC. There are five groups: clones between two functions are called *Function to Function Clones*; clones between two programming structure regions such as unions, enumerators and structs are called *Structure Clones*; clones between macros are called *Macro Clones*; clones between two regions of different types are classified as *Heterogeneous Clones*; clones between external variable definitions, prototypes, and type defines are considered *Misc. Clones*. For example, given a RGC, region 1 may be a function, and region 2 may be a macro. In this case we have a *Heterogeneous Clone*.

3.3.3. Function to Function Clones. *Function to Function Clones* are further divided into four subtypes: *Function Clones, Partial Function Clones, Cloned Function Body* and *Clone Blocks*. *Function Clones* are functions that share a minimum of 60% of there code. *Partial Function Clones* are near miss function clones, where typically one function tends to be a slightly extended copy of the other. These function pairs must have one function with at least 40% of its code shared with the other, and a maximum of 60%, and the other function must have a minimum of 60% shared code. *Cloned Function Body Clones* are functions where one smaller function has been copied into a considerably larger one. This class of clones requires one function to share a minimum of 60% of its code with a function where the shared code only makes up a maximum of 40% of the function code. *Clone blocks* are blocks of code that are not large enough or numerous enough to be in one of the above clones. At this point, clones are individually classi-

[1]We are currently working on are fi ltering switch statements and long lists of if/else statements. These types of code are often false positives.

fied within their RGC. This group is further subdivided as shown below.

3.3.4. Clone Blocks. *Clone Blocks* can exist in various spots in a function and can have many different roles. This group has been further subdivided into nine groups of clones: clones that occur at the beginning of functions are called *Initialization Clones*; clones occurring at the end of functions are called *Finalization Clones*; clones of loops are called *loop Clones*; clones occurring in switch statements called *Clones in Switch*; clones of *if-statements* are called *Conditional Clones*; clones of several conditionals are called *Multi-Conditional Clones*; clones where the body of a conditional has been cloned somewhere else are called *Partial Match Conditionals*. For clones which cannot be easily placed we put them in a group are called *Unclassified*.

Initialization Clones start in the first 5 lines of a function and finish before the middle of the function. *Finalization Clones* start after the middle of the function and end within the last 5 lines. For this group we have found that restricting only one clone to fit this criteria is still accurate but allows for functions with labeled sections of code appended to the end.

Loop Clones are any clones of *for*, *do/while* and *while* loops. These clones must cover at least 60% of the length of the loop and the loop must cover at least 60% of the clone. This clone type allows multiple loops to be part of the clone, as long as 60% of the clone is covered by loops and 60% of the total loops are covered.

While originally part of *Conditional Clones*, *Clones in Switch* have been categorized on their own in this taxonomy. Clones such as these tend to have a high number of false positives because of the rather common way in which *switch-statements* are structured. We are currently working on intelligent filters for this group of clones.

Conditional Clones clones cover *if/else* and *switch* control flow statements. These clones must cover at least 60% of the body of the control flow statement and the body must cover at least 60% of the clone. In the case of *if/else* statements, *if* and *else* blocks can be separate or together as one statement. If a clone covers both it must cover 60% of both statements together. *Multi-Conditional Clones* are similar to the previous clone but allows multiple control-flow statements to be involved. This is useful when regions of many single *if* statements are strung together and cloned. *Partial Match Conditionals* are similar to *Cloned Function Body Clones*. They are clones where the body of a control flow statement has been copied and used in a region that is not a conditional or when one smaller control-flow statement is copied into a larger one. Clones of this type must have one clone covering 60% of a conditional and the conditional must cover 60% of the clone. Again, the thresholds chosen were those that appeared to work well in practice.

3.3.5. Structure Clones, Macro Clones, Misc. Clones, and Heterogeneous Clones. These groups of clones are sub-grouped as *Identical*, *Nearly Identical* , *Similar* and *Structs with Cloning*. *Identical Clones* are structural entities that are 100% duplicates of each other. *Nearly Identical* are 80% duplicates, and *Similar* are at least 60% duplicated. *Structs with Cloning* are structures that had clones that were more than 60% identical, but the regions as a whole have less than 60% duplication. A further restriction on these clones is that ignoring white space and comments, the matched lines must be exact matches. Parameterized matches on such structures creates too many false matches because of the nature of the program structure. For example, the following two structure would produce clones:

```
struct {                struct {
  int a;                  char ch1;
  int b;                  char ch;
  float f                 int  clock;
  char c;                 float elapsed;
  char* charstar;         float* intervals;
} struct1;              } struct2;
```

4. Case Study Results

In this section we summarize some of the results of our case studies, as presented in Table 1. We begin by describing the subjects of our case study and then discuss our observations.

4.1. Study Subjects

In this study, we analyzed the clones of two software systems, the Linux kernel file-system subsystem version 2.4.19 and Postgresql 7.4.2.

The Linux kernel file-system subsystem is the portion of the Linux kernel that supports file I/O. It implements support for 42 file system formats that are managed through a Virtual Filesystem Switch. It consists of 280177 LOC with 537 source files.

Postgresql is an advanced database system that started in 1986 at the University of California at Berkeley and has since become one of the most advanced open source database systems in development today. It also have been written in C and consists of 543387 LOC and 1097 source files.

4.2. Classification of Clones

A summary of the clone classification of our study subjects can be seen in Table 1. In column 1 we see the different types of clones. (Clone types that we did not see in the presented case studies are omitted.) We have hidden the subtypes of non-*Function to Function clones* for brevity. In the columns to the right, we see the number of pairs of regions (such as functions and unions) that contained clones of that type, and the number of clone pairs that were part of those clone types. For example, in the Linux kernel file-system subsystem, in the same directory, there are 216 function pairs that are clones, and 489 clone pairs make up those function clone pairs. In this case, several of the *Function Clones* are composed of several smaller segments of cloned code.

Using the struct filters, we were able to filter 1439 clones of 5336 from the file-system results. That is 27% of the clones reported. This significant improvement is beneficial in both exploration of clones and measuring accuracy of results. In Postgresql, the numbers were even more impressive, we removed 72260 clones from 116036 clone pairs, or 62% of the reported clone pairs. Many of these clones were false matches between structs and other region types. Also, there were many false matches between prototypes, caused by the use of macros and also caused by the large number of single parameter prototypes. This was an enormous improvement with obvious benefits.

From Table 1 it can be seen that we were able to classify a large percentage of the clones in the software systems. Of the remaining 3897 clones in the Linux kernel file-system, we were able to classify 85% of the clones. In Postgresql, only 13% of the clones remained unclassified. Even as such, the groups that remained unclassified are of a non-trivial size, and are the subject of future work.

After filtering, clones that occur outside of functions are somewhat rare. Here we see an immediate benefit of clone classification. From both case studies, we can see that cloning not involving functions makes up less than 1% of the total clone pairs. If the clones were not categorized, such clones would be lost in the vast amount of false positives.

There tends to be fewer unclassified clones in *Same Directory Clones*. This phenomenon is caused by developers' tendencies to duplicate functions rather than fragments of code when in the same directory. There is a lower percentage of *Function Clones* in *Different Directory Clones*. This is caused by the larger differences in requirements and purpose for code in different directories than code in the same directories. In fact, in both studies we see a higher ratio of *Partial Function Clones* and *Cloned Function Body* to *Function Clones* in the different directory group than other groups. Our hypothesis is that functions are cloned from other directories, but are often significantly changed to fit the new requirements of the new subsystem, making them difficult to detect. Additionally, evolution likely causes further gaps in their similarity.

4.3. Cloning Activity

4.3.1. Overall Cloning Through Out the System. As noted earlier, cloning within the same directory but in different files is often manifested in clones of functions or most of a function. In both case studies, we see that at least 79% of the clones in the same directory but different files are either in *Function* or *Partial Function Clones*. When cloning happens in different directories, as few as 20% of the clone pairs contribute to *Function* or *Partial Function Clones*. From this observation, we see that clone detection techniques that are limited to finding only function clones may be missing a significant portion of the clones in a software system, especially with respect to finding clones across subsystems.

In the case studies, 60% and 81% of the total cloning activity occurs within the same directory in regards to number of clone pairs, either in the same file or in different files. This result seems significant. Perhaps problems such as bug fixing are mostly restricted to same directory. We conceived of two hypothesis to explain this. One is that developers tend to duplicate code from files within subsystems they are working in and that developers are more likely to work in a single directory. This could be for a variety of reasons: perhaps the developer is more familiar with this code; or perhaps it is just simply that more code pertaining to the problem the developer is trying to solve is in the same directory. Another reason why there is more cloning in the same directory rather than in other directories may be that clones just might be harder to find when they are in other directories. The same reason that may make function clones harder to find in different directories could also cause difficulties when detecting clones in general.

4.3.2. Frequency of Clone Types. There were several unexpected results of this study. The first result was the relatively low number of *Loop Clones*. Based on initial surveys of the clones, we expected there to be many more clones in this group. It seems that loops on their own are very rarely copied, at least in these two case studies.

One result we did expect was the large number of clones involving conditionals. While they only make up for approximately 5% of the clone pairs, they are the largest group of clones in the clone blocks sub-category. Clones in this group tends to be quite interesting and often the logic they implement is complex. We would expect that maintainers would be interested in eliminating many of these clones as conditionals can often be hard to debug.

From Table 1, we can see that cloning is very often done with function clones, 38% of the clone pairs in the Linux kernel file-system case study contribute to *Function Clones* and *Partial Function Clones*, and in Postgresql 53% of the clone pairs were part of *function* or *Partial Function Clones*. To further understand this large group of clones, we will need to classify them further, perhaps using the function clone classification suggested in [6, 7]. Questions we wish to address are how similar do these clones tend to be? What types of changes are made?

5. Discussion

5.1. Cloning Comprehension

As a result of the classification of the clones by the taxonomy, the results of clone detection tools can be simplified and better interpreted. Clone detection tools often only report the clones that they have found and provide some browsing capabilities through a simple GUI. Some tools such as `GeminiE` [23] for `CCFinder`, generates metrics for files and clones to aid in understanding the cloning activity in a software system. Our classification can help to see characteristics of cloning in software. For example, we saw that more function related cloning occurred in *Same Directory Clones* in both the Linux kernel file-system and Postgresql. We also saw that the percentage of *Function Clones* drops in *Different Directory Clones*. Because we are given the opportunity to observe these characteristics, we are now given new avenues of research. Now we can try to answer the questions of why reality is as it is.

Also, because we are grouping clones by region, users of clone detection tools have yet another metric to use when evaluating cloning in a software system. Grouping clones by region allows the user to see how many regions have cloning between each other, and also how many regions in total have cloning activity in them. These metrics are useful when determining the extent of cloning from a "number of sites infected" point of view, or how many regions are cloned to a certain degree.

5.2. GUI Support

We have begun to incorporate our taxonomy into a GUI as shown in Figure 1. On the left side of the interface, the taxonomy is represented as a tree. Clones are sorted by type and can be navigated to easily through the tree. Clones are grouped by the region pairs they exist in, so the user can select the region pair and see all the clones between the two regions highlighted in gray, or they can select an individual clone which will be highlighted in yellow, and all other clones of the same type in the region pair will be highlighted gray. Clones that are in the RGC but are not of the same type are highlighted in blue. This helps the user grasp the context of the clone that they are viewing.

In Figure 1 we see that a *Conditional Clone* has been selected. The highlighted regions on the right top and right bottom are the two clone segments which are cloned.

By grouping the clones by region, the user has a more compressed view of cloning in the software system. Rather than viewing each clone one at a time, users will see all clones involved in a function, and more quickly understand the significance of the clones and how to deal with them.

5.3. Filtering

Filtering of clones can improve the precision of a clone detection method, ultimately leading to greater usability of clone management systems. Categorizing clones provides a unique way of filtering. Different clone types can be prone to different kinds of false positives. Filtering classified clones allows us to take advantage of this fact. As seen in the case study, significant reductions in the number of reported clone pairs can be made using the regional information of the source code. In Postgresql we were able to eliminate up to 65% of the clone pairs reported. This leads to a more manageable set of duplicated code.

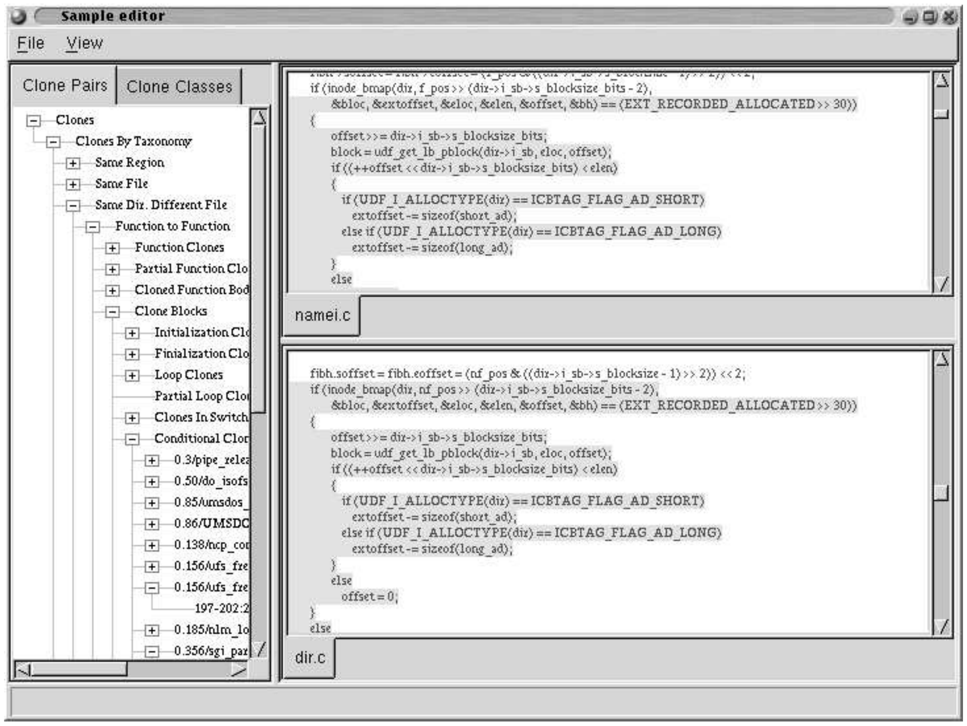
6. Related Work

There are several types of clone detection techniques that have been developed. Metrics-based clone detection tools which detect clones of full blocks of code such as functions based on various metrics extracted from them have been developed by Mayrand et al. [22] and Kontogiannis et al. [20]. Parameterized string matching is discussed by Baker et al. [3, 4] and Kamiya et al. [15]. Baxter et al. [8] have developed a clone detection tool by performing subtree matching on abstract syntax trees. Program dependence graphs have been used by Krinke et al. [21] and Komondoor et al. [18] in detecting duplicated code. Johnson [13, 14] proposed using a fingerprinting algorithm on substrings of the source code. Kontogiannis et al. define two other methods to detect clones in [20]:dynamic pattern matching which finds the best alignment between two code fragments, and statistical matching between abstract code descriptions patterns and source code. Balazinska et al. [5, 7, 6] uses metrics based clone detection to quickly find candidate clones and uses an algorithm based on Kontaogiannis et al.'s dynamic pattern matching algorithm.

Clone detection case studies on the Linux kernel have been reported in [2, 9, 11]. In [9], Casazza et al. use metrics based clone detection to detect cloned functions within the Linux kernel. They performed analysis across the major subsystems, and then on the architecture dependent code of the memory management subsystem and the kernel core. The conclusions of this study were that in general the addition of similar subsystems was done through code reuse rather than code cloning, and more recently introduced subsystems tended to have more cloning activity. Antoniol et al. [2] did a similar study, evaluating the evolution of

	Linux fs		Postgresql	
Type	**Num. Region Pairs**	**Num. Clone Pairs**	**Num. Region Pairs**	**Num. Clone Pairs**
Same Region	**258**	**939**	**733**	**4720**
Function Region	256	935	712	4459
Macro Region	2	4	21	261
Same File	**743**	**1278**	**4997**	**9819**
Function to Function	*741*	*1275*	*4977*	*9719*
Function Clones	214	493	2235	5368
Partial Function Clones	39	59	1038	1534
Cloned Function Body	11	16	285	537
Clone Blocks	477	707	1419	2309
Initialization Clones	63	63	225	227
Finalization Clones	104	104	266	292
Loop Clones	8	9	26	34
Clones In Switch	4	5	47	167
Conditional Clones	34	36	314	407
Multi-Conditional Clones	54	57	96	108
Partial Match Conditionals	47	47	105	120
Unclassified	225	367	537	935
Macro To Macro	0	0	1	2
Heterogeneous	1	1	19	19
Same Dir. Different File	**789**	**989**	**2828**	**12731**
Function to Function	*789*	*989*	*2753*	*12646*
Function Clones	663	709	1158	7163
Partial Function Clones	19	78	482	7163
Cloned Function Body	8	8	127	737
Clone Blocks	99	194	986	2120
Initialization Clones	9	9	75	83
Finalization Clones	14	14	187	269
Loop Clones	1	1	15	16
Clones In Switch	1	1	18	23
Conditional Clones	12	14	70	211
Multi-Conditional Clones	13	37	95	117
Partial Match Conditionals	10	14	144	195
Unclassified	43	104	541	1191
Programming Structs	0	0	*2*	*2*
Macro To Macro	0	0	*10*	*20*
Heterogeneous	0	0	*1*	*1*
Misc.	0	0	*5*	*5*
Different Directory	**384**	**702**	**7097**	**16506**
Function to Function	*373*	*666*	*6948*	*16329*
Function Clones	123	174	2756	5946
Partial Function Clones	15	55	112	716
Cloned Function Body	18	92	469	4266
Clone Blocks	217	345	3611	5401
Initialization Clones	14	14	305	353
Finalization Clones	58	63	298	703
Loop Clones	3	45	6	13
Clones In Switch	4	45	85	188
Conditional Clones	26	32	102	114
Multi-Conditional Clones	36	37	141	254
Partial Match Conditionals	37	40	376	436
Unclassified	53	101	2416	3316
Programming Structs	*2*	*2*	*35*	*63*
Heterogeneous	*9*	*34*	*8*	*8*
Misc.	*0*	*0*	*106*	*106*

Table 1. Frequency of various clone categories — Parametric String Match



```
if (inode_bmap(dir, f_pos >> (dir->i_sb->s_blocksize_bits - 2),
    &bloc, &extoffset, &eloc, &elen, &offset, &bh) == (EXT_RECORDED_ALLOCATED >> 30))
{
    offset >>= dir->i_sb->s_blocksize_bits;
    block = udf_get_lb_pblock(dir->i_sb, eloc, offset);
    if ((++offset << dir->i_sb->s_blocksize_bits) < elen)
    {
      if (UDF_I_ALLOCTYPE(dir) == ICBTAG_FLAG_AD_SHORT)
        extoffset -= sizeof(short_ad);
      else if (UDF_I_ALLOCTYPE(dir) == ICBTAG_FLAG_AD_LONG)
        extoffset -= sizeof(long_ad);
    }
    else
```

```
fibh.soffset = fibh.eoffset = (nf_pos & ((dir->i_sb->s_blocksize - 1) >> 2)) << 2;
if (inode_bmap(dir, nf_pos >> (dir->i_sb->s_blocksize_bits - 2),
    &bloc, &extoffset, &eloc, &elen, &offset, &bh) == (EXT_RECORDED_ALLOCATED >> 30))
{
    offset >>= dir->i_sb->s_blocksize_bits;
    block = udf_get_lb_pblock(dir->i_sb, eloc, offset);
    if ((++offset << dir->i_sb->s_blocksize_bits) < elen)
    {
      if (UDF_I_ALLOCTYPE(dir) == ICBTAG_FLAG_AD_SHORT)
        extoffset -= sizeof(short_ad);
      else if (UDF_I_ALLOCTYPE(dir) == ICBTAG_FLAG_AD_LONG)
        extoffset -= sizeof(long_ad);
    }
    else
      offset = 0;
}
else
```

Figure 1. CloneNavigator Screenshot

code cloning in the Linux, concluding that the structure of the Linux kernel did not appear to be degrading due to code cloning activity. In [12] a preliminary investigation of cloning among Linux SCSI drivers was performed.

Kamiya et al. [15] performed tests on JDK to search for clones within the system, and they studied the cloning behavior between Linux, FreeBSD, and NetBSD. While [15] observes that clones in JDK seem to occur in near directories or files based on visual inspection of the scatter plot their tool presents, no quantitative data analysis is discussed concerning this point. None of the above studies have discussed the types of clones they have found, or discussed the locality of code cloning in detail other than comparing the level of cloning amongst subsystems.

A work similar to this also tries to categorize clones for the purpose of software maintenance. In [5], Balazinska et al. create a schema for classifying various cloned methods based on the differences between the two functions which are cloned. The results produced in [5] are used by Balazinska et al. in [6, 7] to produce software aided re-engineering systems for code clone elimination. This differs from our work in that our classification scheme is based on locality as well as clone type, and copied functions are only one type in our case, although in [5] they break this down into 18 categories. One of our main research goals is to determine how much developers clone and from where. This question is not answered by the clone classification scheme in [5]. Additionally, these 18 categories will become subtypes of *Function Clones* in our taxonomy. In addition, this work ignores code clones which are not function clones.

7. Summary and Conclusions

Management of duplicated code can contribute to the healthy growth and evolution of software systems. Results from clone detection tools are often numerous, and without a form of classification, unmanageable. As a solution we classify code clones according to a taxonomy that separates clone based on locality and functionality. We were able to classify the majority of clones reported by the clone detection tool. We were also able to significantly improve the precision of the clone detection tool by filtering clones based on regional information.

In our two candidate systems, developers tended to clone from within the same directory. This result agrees with the results reported in [15, 17]. Within our candidate software systems, we have found that a large portion of clones contribute to function clones, as many as 50%. We also found that function cloning across directories is either less prominent or harder to detect, leading us to believe that clone detection methods limited to finding only function clones will suffer with regards to recall.

Our approach had positive results in terms of filtering false matches. While in this case study we applied our approach to only parameterized string matching methods, we feel that filtering and categorization with benefit all clone detection techniques. The gains in comprehension via classification are universal and we will apply this tool to results from other clone detection methods. Filtering will most definitely benefit any tools that do not filter already.

In our future work, we intend to include a deeper classification of function clones. We also intend to add to our clone blocks categories in a effort to categorize all clones in the software system. We are currently working on a clone exploration utility that will be a prototype for an IDE plug-in. We are also interested in monitoring cloning over time and how clones change with respect to each other.

References

[1] Exuberant ctags. `"http://ctags.sourceforge.net/"`.

[2] G. Antoniol, U. Villano, E. Merlo, , and M. D. Penta. Analyzing cloning evolution in the linux kernel. In *Information and Software Technology 44(13)*, 2002.

[3] B. Baker. A program for identifying duplicated code. In *Proceedings of Computing Science and Statistics: 24th Symp. Interface*, pages 49–57, 1992.

[4] B. Baker. On finding duplication and near-duplication in large software system, 1995.

[5] 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*, pages 292–303, 1999.

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *The Proceedings of the 6th. Working Conference on Reverse Engineering*, pages 326–336, 1999.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings of the 7th. Working Conference on Reverse Engineering*, pages 98–107, 2000.

[8] I. D. Baxter, A. Yahin, L. M. D. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[9] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Identifying clones in the linux kernel. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 92–100. IEEE Computer Society Press, 2001.

[10] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, 1999.

[11] M. W. Godfrey, D. Svetinovic, and Q. Tu. Evolution, growth, and cloning in Linux: A case study. A presentation at the 2000 CASCON workshop on 'Detecting duplicated and near duplicated structures in largs software systems: Methods and applications', on November 16, 2000, chaired by Ettore Merlo; available at `http://plg.uwaterloo.ca/˜migod/ papers /cascon00-linuxcloning.pdf`.

[12] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the 2000 International Conference on Software Maintenance*, 2000.

[13] J. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of CASCON 93*, pages 171–183, 1993.

[14] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintanence*, pages 120–126, 1994.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. In *Transactions on Software Engineering 8(7)*, pages 654–670. IEEE Computer Society Press, 2002.

[16] C. Kapser and M. W. Godfrey. A taxonomy of clones in source code: The re-engineers most wanted list. In *2nd International Workshop on Detection of Software Clones (IWDSC-03)*, 2003.

[17] C. Kapser and M. W. Godfrey. Toward a taxonomy of clones in source code: A case study. In *Evolution of Large Scale Industrial Software Architectures*, 2003.

[18] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–??, 2001.

[19] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of Working Conference on Reverse Engineering*, pages 44–55. IEEE Computer Society Press, 1997.

[20] K. Kontogiannis, R. D. Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection, 1996.

[21] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[22] 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 International Conference on Software Maintenance*, pages 244–253. IEEE Computer Society Press, 1996.

[23] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the Eighth IEEE Symposium on Software Metrics*, pages 67–76. IEEE Computer Society Press, 2002.