JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2005; **00**:1–10 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

Supporting the Analysis of Clones in Software Systems: A Case Study

Cory J. Kapser[1] and Michael W. Godfrey[1]

[1] *David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1*

KEY WORDS: Code clone, duplication, software architecture

SUMMARY

Code duplication is a well documented problem in industrial software systems. There has been considerable research into techniques for detecting duplication in software, and there are several effective tools to perform this task. However, there have been few detailed qualitative studies into how cloning actually manifests itself within software systems. This is primarily due to the large result sets that many clone detection tools return; these result sets are very difficult to manage without complementary tool support that can scale to the size of the problem, and this kind of support does not currently exist. In this paper we present an in-depth case study of cloning in a large software system that is in wide use, the Apache web server; we provide insights into cloning as it exists in this system, and we demonstrate techniques to manage and make effective use of the large result sets of clone detection tools. In our case study, we found several interesting types of cloning occurrences, such as "cloning hotspots", where a single subsystem comprising only 17% of the system code contained 38.8% of the clones. We also found several examples of cloning behavior that were beneficial to the development of the system, in particular cloning as a way to add experimental functionality.

Introduction

Code duplication, or code cloning, is generally believed to be common in large industrial systems [5, 8, 11, 18, 23, 24, 26]. Management of code cloning and the various problems associated with it is important for the successful evolution of software systems. An example of how problems can arise is when multiple copies of one piece of code must be modified to fix a single bug. This leads to wasted effort in both finding and fixing the clones. To ensure the successful evolution of large software systems, such problems must be addressed.

One problem of code clone analysis is that clone detection tools may return a large set of suspected clones, but provide little or no additional information about them to aid the user in their interpretation. This makes clone analysis cumbersome at best and intractable in general. Viewing and classifying thousands of clones manually is time consuming and impractical, but it is necessary if one hopes to manage clones successfully; for example, in our Apache httpd case study, a naive use of the clone detection tool CCFinder resulted in 13,062 clone pairs. As a result, little work has been done concerning the in-depth investigation of cloning as it occurs in software systems. This paper presents such an in-depth investigation using a tool designed to help overcome these problems.

The goal of this work is to provide a more detailed view of source code duplication within real software systems as well as describe the criteria for an effective clone analysis tool. Through a case study on the Apache httpd web server, a software system of non trivial size that is in wide use, this paper provides several insights into how software developers duplicate code and also how this duplication manifests itself within the software system, partially confirming and partially contrasting previously reported results. In the case study, we describe "hotspots" of cloning activity, and examples of good kinds of cloning. We also describe the types of clones that occur, and where they tend to be within the system.

The case study also provides insights into clone detection and analysis of large software systems. It presents a preliminary set of requirements for an effective clone navigation and visualization tool. We describe an approach to managing and organizing large clone sets to better aid the developer in the task of clone analysis, and several possible ways in which additional information about the software system, such as system structure, can be used as part of the analysis process.

In the paper that follows, some background on code cloning is presented. Then an idealized description of a tool that supports the analysis of code duplication is presented, followed by the description of a prototype tool that has been designed to meet these criteria. Cloning in the Apache httpd web server is analyzed in detail and the important features of the tool are discussed insofar as they pertain to the case study. Finally, a summary of the case study highlights important observations about cloning in Apache, and these findings are related to previous work.

Code Cloning

Code cloning is considered a serious problem in industrial software [4, 10, 5, 8, 11, 15, 18, 23, 24, 26]. It is suspected that 5 to 10% of many large systems is duplicated code [5, 11], and it has been documented to exist at rates of over 50% in a particular COBOL system [11]. The literature on the topic has described many situations that can lead to the duplication of code within a software system [5, 8, 15, 18, 24, 26]. Many of these can be considered ill intentioned cloning. For example, developers may duplicate code because the short term cost of forming the proper abstractions may outweigh the cost of duplicating code. Developers may also duplicate code when they do not fully understand the problem, or the solution, but they are aware of code that can provide some or all of the required functionality. Clones can also be introduced as a side effect of programmers' memories; programmers may repeat a common solution, unknowingly introducing clones into the software system [8].

Duplicates can also be introduced with good intentions. Duplicating code can, in some situations, be used to keep software architectures clean and understandable. Duplicates can also be used to keep unreadable, complicated behavior from entering the system. Finally, lack of expressiveness of a given programming language may lead to the use of "boilerplated" solutions for particular problems, or even source code generation. This kind of technique is common in COBOL development, for example. In these cases, the use of cloning is typically well understood by the developers, and the aim is to prevent errors by re-using trusted solutions in new contexts.

Whatever the intention of the developers, the practice of code duplication can have a negative impact of the stability and maintainability of a software system. For example, the size of the source code base, and ultimately the size of the object code, may become significantly larger as a result of excessive code cloning [5, 15]. 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 [15]. 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. Copying code may also result in copying bugs within the code as well.

A Tool To Explore Clones

In this section we briefly outline our general criteria for a tool used to navigate and understand cloning in a software system. We then describe in more detail the features needed to meet these criteria. This set of criteria and features is derived from much manual work by the authors in attempting to understanding cloning in software systems, starting with our studies of the Linux kernel file-system subsystem and the database server Postgresql [19, 20]. Some features were also taken from suggestions by students in a senior level graduate course who used the tool Gemini [28] and CLICS, the tool described in this paper, to perform an analysis of clones within the Linux kernel source code.

Criteria

The core challenge to the maintenance and management of cloning in software systems is comprehending the actual types of clones and the dependencies they create within the software system. To complete such a task, the duplication must first be detected, and then evaluated throughout the system at different levels of abstraction. Methods to find the duplication in the source code has been a topic of research for some time now and there are many techniques to detect clones [5, 7, 8, 11, 15, 18, 24, 25, 26]. However, clone detection tools can return very large result sets and viewing every possible clone is generally infeasible. To address this problem, tools and processes need to be developed to help guide the software maintainer toward the information they require to complete the task. We consider that any tool designed to help navigate and understand cloning in a software system should provide:

1. facilities to evaluate the overall cloning situation,
2. mechanisms to guide users toward clones that are most relevant to their task, and
3. methods for filtering and refining the analysis of the clones.

Each of these criteria is described in more detail below.

Overall System Evaluation

As a first step in understanding cloning within a software system, regardless of the end goal, maintainers must have an understanding of the cloning from a high level of abstraction. This understanding will allow the user to evaluate the extent and the severity of the duplication in order to estimate cost and/or necessity of the task. This information will also be a starting point to guide them through the rest of their task.

Several mechanisms can be used to evaluate cloning from a high level. Visualization methods, such as scatter-plots [5, 11, 18, 27, 28], are useful for the discovery of highly related sub-systems and high levels of cloning within a subsystem. They are also useful for detecting unusual types of cloning, such as cloning from system libraries to other parts of the software system. Metric–oriented reports, such as reporting the percent of lines cloned, average length of clone, *etc.* are useful for directing users to points in the system where the most cloning is occurring, or where cloning activities are unusually high in relation to subsystem size. Whether the method is visualization or metric oriented, it is important that the information provided be scalable and covers a wide range of properties of the system and its clones.

Guide and Empower the User

The possibly large sets of clones returned by the clone detection methods make it infeasible to look at each individual clone. A tool designed to aid in the comprehension of cloning in a software system should provide a means of guiding the user to clones relevant to his/her task and reduce the data set as much as possible without loss of relevant information.

There are several ways to direct users toward the clones they seek. Metrics can be used to query the data set [14]. Some examples of metrics that might be used are the size of the clone, the types of changes made to the clone, and types of external dependencies a code segment has. Such a method can direct users to promising refactoring opportunities. Other methods of querying the data set can also be used, such as querying based on location of the clones in the software and the type of source code entity the clone exists. For example, a user might be concerned about cloning of macros originating in a particular file. Querying mechanisms provide flexible analysis, allowing users to leverage their own knowledge about the software and cloning, making the user more effective in their task.

Upon initial survey of a software system, users may not be fully aware of what information they want or need. Query facilities can suffer from this weakness and strong static analysis of the data set is also required. Static analysis should provide low level metrics about cloning activities in the system. Additionally, the tool should provide a method of navigating through clones that leverages general knowledge about cloning. An example of this would be categorization of the clones as is done in [7, 20]. This will provide a method of education for novice users, and guide experts more quickly to clones relevant to their task.

It is important to provide views that describe cloning in terms of the concrete architecture. We believe that relating cloning and architecture can have great benefits to comprehension of cloning. Cloning is a type of implicit architectural dependency, and as such can provide information about the high-level design of the system. This also enables users to use their own knowledge of the architecture of the system when evaluating clones.

Analysis Refinement

Due to the subjective nature of the analysis of clones, from the perspective of the user, there will always be clones that do not apply to their task, For this reason, it is important that tools supporting the comprehension of cloning provide mechanisms to remove and filter clones from the analysis.

The CLICS Tool

The *CLone Interpretation and Navigation System (CLICS)* is a prototype implementation of a tool that meets the criteria listed above. This section will describe the features we implemented in an effort to meet the criteria for an effective clone analysis tool. These features represent alternatives that could be used to meet the criteria but are not presented as the only way to do this.

System Overview

CLICS was first described in [20]. Currently, CLICS is configured to use clone detection results from the tool CCFinder [21] as the dataset, although in principle it could be adapted to use the output from almost any clone detection tool. CLICS uses a taxonomy of clone types to categorize clones and

System	Total Clones	Simple Call Filter	Overlap Filter	Logical-Structures Filter	Non-Function Filter
Apache	13,062	6,795	722	372	192
gnumeric	14,816	1,215	1,092	3,223	1,565
postgres	144,665	15,035	3,369	53,001	36,358

Table I. Number of clone pairs removed per filter

generate statistics about clones in a software systems. The taxonomy, first described in [20], has been constructed through manual classification of clones in several software systems. Currently, CLICS supports clone analysis in C/C++ and Java. While we have used this tool to analyze object oriented systems in other case studies, the taxonomy is currently very procedurally oriented. Class level clones need to be taken into account in future versions, as well as the additional relationship types such as inheritance.

Extracting Regions from Source Code

In our first step we invoke ctags, a tool for extracting indices of language entities found in the source code. Because ctags only finds the start of a software entity, we use a script to 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. Using this information we map the file to eight types of regions: consecutive type definitions, prototypes, and variables; individual macros, structs, unions, enumerations, and functions. Comments are ignored in the analysis.We also extract subregions of functions, including initialization of variables, code blocks such as control flow statements, and sequences of function calls.

Mapping Clone Pairs to Regions

Next, 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 breaks clones that are part of a function at the end of that 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 up the clone, but in practice we find the current method yields acceptable results.

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 example of such a group would be several clones between two functions. The concept of RGC is useful for both visualizing and filtering clones, as we discuss below.

Filtering

CLICS begins by filtering the dataset of several common types of false positives that are detected by parameterized string matching methods. These filters work by enforcing stricter criteria for a clone match in particular areas of the code. There are currently four filters we have implemented. In all cases, the thresholds were chosen based on extensive experimental trials. These trials consisted of trial and error calibration where for each trial the parameters were set according to the results from the previous trial, until no true clones are found in a large sample of the removed clones. In all cases the filters were tested on several case systems, namely Postgresql 8.0.1, gnumeric 1.2.12, and Apache httpd 2.0.49.

1. **Non-function filter**. This filter operates 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. These lines can be matched in any order.

2. **Simple call filter**. Clones occurring on statements that are "simple function calls" can often contribute to many false positives when using parametric string matching algorithms. Regions of code that are "simple function calls" are sequences of code of the form

   `function_name(token [, token]*).`

   The criterion for a match is that 70% of the function names in either region must be similar. To be similar the edit distance, as computed by the Levenshtein Distance algorithm, must be less than half the length of the shortest of the two function names being compared. This value for a match was determined by examining the edit difference of function calls in true clones. In such cases, we found that the edit distance was always less then 50% of the total length of the function name. During our studies, we found that typical clones of function calls would use mostly the same or similar functions, but did occasionally contain calls to completely unrelated functions. To accommodate this, we adjusted the percentage of function names that must match until true clones were not removed from the dataset.
3. **Logical-structures filter**. We found that clones within simple logical structures such as switch statements are often false positives. To filter clones in these areas, we require that 50% of the lines of code in these areas match. Clones in very simple *if-then-else* blocks are also filtered in this way. Initial values for this percentage match were found by analyzing cloning in these regions, and counting the number of lines that remain unchanged in a true clone. We then tuned the filter by making it less strict until we found no true positives were removed by it in the dataset.
4. **Overlap filter**. Clones whose two segments of code overlap by more than 30% of their length are also removed. This value was determined through observation of overlapping clones, and counting the maximum overlap of true clones. The value was then adjusted through several trials.

Although the filtering mechanisms are simple, they are very effective. These filters on average reduce the clone dataset returned by *CCFinder* by approximately 60% in the case studies that we have performed.

The reader should note in Table I the large difference in the number of clones removed by the *non-function filter* for Postgresql compared to our previous results reported in [20]. This difference is caused by a change in the classification of certain functions that could not be parsed by ctags. In the previous version of the tool, these regions were classified as macros, and therefore filtered using the *non-function filter*. The new version of this tool now classifies these blocks as functions. Many of these functions are implementations of scanners and parsers, composed largely of *switch* blocks that were filtered and removed by the *logical-structures filter*.

Sort Clones into Taxonomy and Display Results

Once we have a set of filtered clones, we sort them into the taxonomy. The clone taxonomy, described in [3, 20], is a hierachical classification that first separates clones by how they span subsystems. Clones at this level are either *Same File*, *Same Directory - Different File*, or *nth Cousin Clones* where *n* is the distance measured by the closest common node in the containment tree of the concrete architecture of the software system. Then RGCs are then partitioned by the type of software entity that comprises them, such as functions, unions, and type definitions. Finally they are categorized by the degree of similarity between the two regions of the RGC. In the case where clones do not cover a large percentage of a function, the type of source code entity is also considered, for example *loop clones* are clones between two loops. The user is then presented with a GUI containing views of clones using trees of hierarchical classification, clone classes, and concrete architecture views.

Meeting the Criteria For a Clone Navigation Tool

The following section describes the features we implemented to meet the criteria we described for a clone comprehension tool. It is a proof–of–concept implementation.

Overall System Evaluation

To provide a general system overview, we compute a series of metrics encompassing several aspects of the system: system size, percentage of system cloning, and frequency of clone types. The metrics detailing system size include the number of files and LOC.

Metrics describing percentage of system cloning include percentage of lines that have a clone, percentage of methods containing a clone, and percentage of files containing a clone. Metrics describing the percentage of lines with clones and the percentage of methods with clones are useful indications of the degree of cloning in the software system. The percentage of files containing clones is a useful metric when determining the clone density in the system.

To describe the frequency of different types of clones in a software system, we list the the number of occurrences of each clone type in the taxonomy. These metrics indicate the types of problems that may be occurring in the system and may indicate the degree of difficulty of the investigation and management.

Guide and Empower the User

CLICS uses several mechanisms to enable the user to perform an in-depth analysis of clones in the system. These mechanisms include visualization of clone relations of subsystems using a hierarchical containment graph, metrics for entities at all levels of architectural abstraction, clone navigation through the taxonomy, clone navigation through the subsystem tree, and query facilities.

To visualize cloning as it relates to the system's architecture we use *LSEdit*, which is part of the architecture recovery toolkit, *SWAGKit*. *LSEdit* is a graph visualization tool that is designed for the exploration of software "landscapes", which are graphs that represent software architectures and their dependencies. The nodes of the graphs are software artifacts such as subsystems, files, and methods, and the edges of the graph are dependencies between two software artifacts, in this case clones. Graph entities can be hierarchically contained, allowing varying levels of abstraction during analysis. We visualize the clones in this way because we believe it is more scalable than scatter-plots used in [5, 11, 28], and because we have extensive experience in using this visualization tool to perform architectural modelling of many large software systems [9].

Using the clone taxonomy described earlier, users can explore the clones in the system by type. This view uses a *clone type navigation tree*. Users can view clones in each category, and remove any clones they believe to be false positives. This method of navigation is especially useful when performing the initial analysis of clones in the system. It can provide insight into what types clones are most frequent within the system, and inexperienced users can use this navigation method as a way to become familiar with the clone classifications. This navigation tree is also used to sort the results from the queries described below.

Complementary to the LSEdit visualization, navigation through the system architecture can also be done through the *system navigation tree*, shown in Figure 3(b). This tree is structured to represent the subsystem containment hierarchy of the software. In addition to showing the degree of relationship between subsystems, as shown in the figure, metrics describing the cloning situation within the software entity are also provided. For each software artifact within a selected entity on the tree, the following metric values are shown:

1. number of clones involving only entities within the referenced entity,
2. number of clones with one segment within the referenced entity, and one segment elsewhere in the software system,
3. percentage of lines of code of the software system contained within the given entity,
4. percentage of total clones that are involved with referenced entity,
5. number of RGCs in the referenced entity, and
6. total lines cloned in the referenced entity.

These metrics are useful for providing information about the validity of the clones within a subsystem. For example, a subsystem containing a large number of clones but relatively few RGCs is a good indication of overlapping clones. They also guide the user to points in the system where the most cloning is occurring.

Currently only limited query support is implemented in CLICS. CLICS supports querying clones based on location, based on clones relations to code segments, and clones of a particular size. Queries of clones based on location include clones strictly within a given entity, clones going from one entity to another, and clones that have at least one of its code segments in the entity.

Querying for clones related to a region of code includes queries for clones that are directly related to the code, and for clones that are related transitively. Transitively related clones are found by computing the transitive closure of a generalized clone relation. Such queries often uncover clone relations that are too different to be detected by regular clone detection methods.

Analysis Refinement

Refinement facilities in CLICS currently allowed to the manual removal of clones and removal/addition of files from the analysis. Users can remove individual clones, whole RGCs, and clone classes. They can also select files to be excluded from the analysis. More advanced filtering mechanisms are currently being developed.

Case Study

In this section we will describe the case study we performed on the Apache httpd web server [1]. Based on NCSA httpd 1.3, the first release of Apache was made in April 1995. Since then, it has become tremendously popular. As of March 2005, more than 68% of web sites are served by Apache [2]. Apache is a medium sized software system. Our study focused on Apache version 2.0.49, which consists of 709 *.c* and *.h* files and 261,219 LOC. The architectural representation we use to model *Apache* was derived directly from the directory structure of the source code.

We chose Apache for our case study because it is a system in wide use that is of non trivial size, and has several interesting characteristics in its architecture that we wished to investigate. The first characteristic is that it runs on several different platforms: BeOS, Netware, OS/2, Unix, and MS-Windows. In particular, there is a Multi-Processing Module (MPM) for each of these platforms. It also

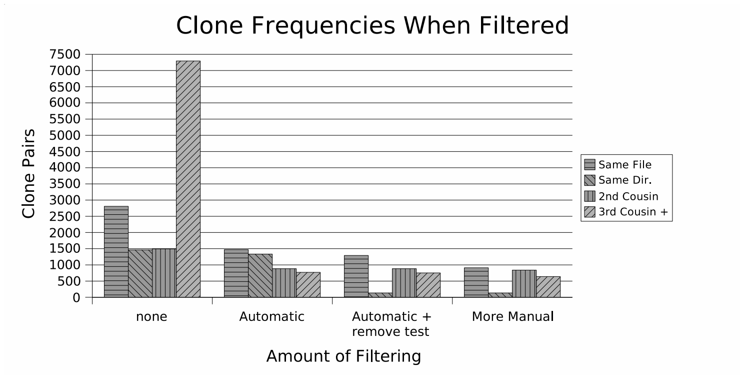


Figure 1. Frequency of clones at different levels of filtering

has several experimental MPMs released with the source. These modules are responsible for process management and were expected to have a high degree of cloning amongst them as they implement very similar functionality.

Through this case study, we evaluated the value of many of these features at different levels of the analysis. We found that viewing cloning as a dependence relationship in a concrete architecture of a software system provided structure for the analysis and aided the process of separation and investigation of different clones in the system. We also found that metrics are a good tool for initial inspection of the cloning, and also act as a guide to *cloning hotspots* in the system.

Experimental Parameters

Using CCFinder 7.2.4 we detected clones in the source code. CCFinder allows the user to set the minimum size of a clone that will be detected. We chose 30 tokens to be the minimum size of detected clones. This size was chosen after some experimentation and was found to have an acceptable level of accuracy; we found that smaller values led to a large increase in false positives in the data set. No other parameters were set for this case study. All thresholds for categorization and filtering are fixed as described in this paper. Future versions of the tool will provide users with the ability to adjust the taxonomy and the filters.

The Data Set

Using the filters we described earlier, a significant portion of clones were removed from the data set. Figure 1 illustrates the number of clones removed by our filters. Using only automatic filters we were able to remove 8081 clones, 61.8% of the total dataset. The *simple call filter* removed a largest amount, removing 6795 clone pairs. The *overlap filter* removed 722 clone pairs. 372 clone pairs were removed by the *logical-structures filter*, and 192 clone pairs were removed by the *non-function filter*. It should be noted that these values vary from system to system, as can be seen in Table I.

We believe that very few true positives were removed by our filters. We assessed the quality of the filtering through manual analysis of the filtered clones. Following the study reported in [21], we increased the size of our sample, viewing all of the results from the *logical-structure filter*, *non-function filter*, and *overlap filter*. We also examined 500 of the clones filtered by the *simple call filter*. We did not find any case where a true clone was removed. However, statistical significance has not been computed. Because this is a single-user case study done by the tool authors, it is not clear that this result is generalizable.

Figure 1 shows the distribution of clones within the software system at various stages of filtering the results: *no filtering*, *only automatic filtering*, *automatic plus remove test*, and *more manual filtering*. For each stage the group of four bars represents clones that occur in the same file, same directory, between two directories with the same parent, and between directories that are more distant. In the group *no filtering*, the results from *CCFinder* were used directly. In *automatic filtering* the filters described above are used.

In the group *automatic filtering plus remove test* we removed a subsystem from the analysis that was causing a very high number of false positives, namely *apr/test*. We found this subsystem through the investigation of a *cloning "hotspot"*. In our initial investigation noted that 60.8% of the clones resided within the *apr* subsystem. Delving deeper, we found that the *testing* subsystem of *apr* held most of those clones, 32.6% of the total clones. Excluding this subsystem removed an additional 1405 clones from the data set. The subsystem is composed of test suites who use the *C Unit Testing Framework (CuTest)*. *CuTest* uses a very simple and repetitive method of adding tests to a test suite. The vast majority of the clone pairs were between code for building the test suite, all calls to a single function. This call to a single function was the source of the false positives in this subsystem.

In the group *more manual filtering* we removed false positives through analysis of clones across high level subsystems and also by removing clones involving *printf* statements. The more involved filtering was done by evaluating unusual dependencies that appeared in the visualization and removing clones that were clearly false positives.

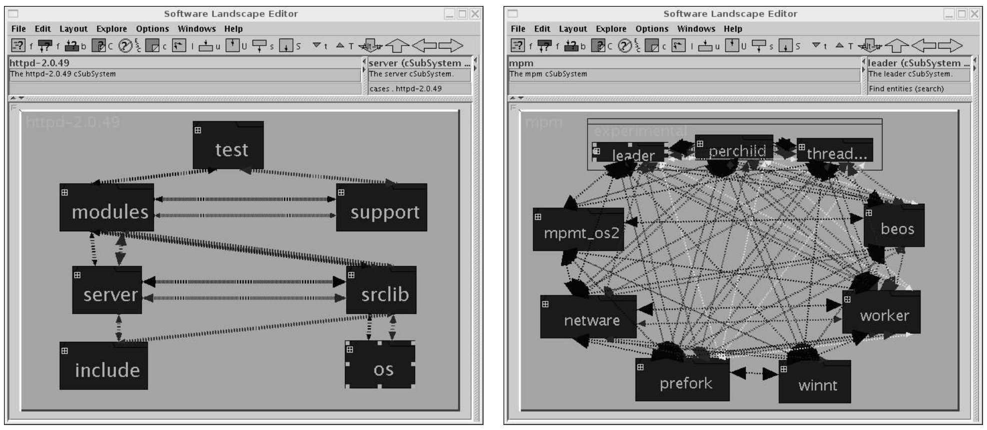
The reader should note that while the number of clone pairs in the data set was greatly reduced, there was only a minor reduction in the actual percentage of the system that contains clones. With no filtering, 15.6% of the system was part of a clone pair. After automatic filtering, 12.7% of the system was involved in a clone pair. After all filtering was done, 12.0% of the system was involved in a clone pair. This is because only small portions of the code generate a large percentage of the false positives.

Cloning - From the High Level

At first glance, Apache exhibits several characteristics that appear to distinguish it from previous reports on other software systems. For example, studies suggest that most cloning tends to occur within the same directory or subsystem [18, 19, 20]; however in Figure 1 we see cloning in Apache is somewhat more prominent across subsystems rather than within the same subsystem. A more detailed break down of the types of clones found in this case study can been seen in [21].

When no manual filtering is done, we found that a high degree of cloning has occurred across subsystems. After filtering, both manually and automatically, we see that 51% of the clones in *Apache* are crossing subsystem boundaries, shown in our taxonomy as the cousin clones. This is explained by the method of behavioral duplication the *Apache* team used when porting *Apache* to the platforms it runs on today. In many cases, when platform specific code was required, such as in the *mpm* subsystem, a substantial amount of code appears to have been copied and then ported to the platform.

Figure 1 illustrates the importance of filtering. There is a drastic difference in the number of *Same Directory Clones* from the initial automatic filtering and the subsequent manual filtering carried out. Two important points can be drawn from this observation. First, one can see the impact of false positives on the results and the importance of careful inspection of the data set before reporting results. Second, it highlights the usefulness of providing metrics for each subsystem as the users explore clones from the architectural perspective. Within five minutes, the authors identified the *apr/test* subsystem anomaly by noting the disproportionate percentage of clones in the subsystem when compared to its relative size.



(a) **(b)**

Figure 2. Clone dependency view in LSEdit.

During our case study we found that 60% of all clone pairs contributed to *function clone* pairs. For clones outside of the same file, 70% of clones contribute to *function clones*. From this set of clones, we found that *function clones* outside of the same file are composed of 2.5 clone pairs on average. This is an interesting result. It indicates that the functions that have been cloned may have several non-trivial changes in the code. This indicates that the functions are likely going to be difficult to be refactored. We also found that there are several non-function clones, although very few. Non-function clones make up less than 3% of the total cloning in the software system. These clones are *Macro clones, Prototype clones*, and *Clones of Global Variables*. These clones tend to be small and unchanged. We believe these clones should be refactored.

Lessons Learned

Immediately from these results we can see the value of providing a variety of views and metrics to facilitate initial high level analysis of the system. At this stage we already know the types of clones that are prevalent in the software system. By looking at the distribution of the clone types in terms of location, we can also see that Apache appears to be reasonably well organized. Clones are generally taken from source code that involves related concepts, and because most of the clones are at least within the same major subsystem it seems that most concepts are clustered together closely.

Using metrics such as the number of clone types and the location of clones can provide an interesting initial view of the software. However, without further investigation we only have a general idea of the types of cloning to expect, and cannot make concrete inferences about the quality of the cloning itself.

A key reason visualization worked well for filters in this situation is that it was tied to an architecture that the authors had some understanding of. This allowed for the use of prior knowledge, both about Apache httpd and software systems in general, to evaluate what was to be expected and what was not. Abstraction is also a key point in this case. Through the use of hierarchical containment and propagating dependencies from children to parent, *LSEdit* enables the user to very quickly see relationships in a scalable way. For example, clones between *support* and *server* were easily spotted and investigated, and removed.

Inspecting the Code

During our initial inspection, we were concerned with the various edges between unrelated subsystems, as pictured in Figure 2(a). In particular, we wanted to determine the causes of the edges, and eliminate any false positives contributing to them. The end result of that investigation led, in addition to the filtering mentioned previously, to the removal of several false positives. The total time spent refining the results and investigating the high level dependencies depicted in Figure 2(a) was under two hours. The reader should note that the authors had some knowledge about the architecture of *Apache httpd* prior to this study. We were familiar with the overall concrete architecture, and knew about the responsibilities of the various subsystems. With the assumption that subsystems with unrelated responsibilities should not have cloning between them, we focussed much of our effort on the investigation of edges breaking this assumption. While it helped us find many false positives quickly, it also uncovered a few surprises, as described below.

A surprising dependency seen in Figure 2(a) is the cloning between *include* and two other subsystems, *server* and *srclib*. A single file, *pcreposix.h*, and several function prototypes have been copied from *include*. In the opinion of the authors such a dependency should not exist, and this is an example of bad cloning. Unnecessary effort is required to keep the header files synchronized. These clones should be eliminated by maintaining only one copy of this file.

Clones between the *srclib* and other subsystems were unexpected. *srclib* is primarily composed of the *Apache Portable Runtime Project (APR)*. It is shipped with the Apache web server source code, it is not directly related to web services. It is designed to provide a consistent, platform independent API to underlying platform specific functionality. Cloning between the *server* and *scrlib* involved cloning of time and date formatting, queue control, and bucket management. It was composed of eight function clones and a few other clone types, most of which were identical clones. We also observed that not only were the names of the functions with cloning very similar, but also those of the files involved in the cloning relationship. For example, *fdqueue.c* and *apr_queue.c*. This we believe was an example of misplaced concepts within the concrete architecture.

os and *srclib* shared a single function clone, again an exact duplicate with only the name of the function and the parameter type were changed. *srclib* and *modules* share clones involving the management of a hash table. There are three *function clones* and a *partial function clone* in this set. While the function clones clearly have the same origin, and are nearly identical, we would not suggest refactoring in this case because they are only a few functions of many that are responsible for managing the hash tables in the two subsystems. A much better approach to management would be to either refactor the code using the *modules* hash to use the one in *apr* or document the functions and maintain them in parallel. These two subsystems also share a small code segment involved with command formatting. In this case we see another example of duplicated concepts.

We were not surprised to see dependencies between *modules* and *server*. Much of the server's http processing code was moved from the *server* to the *modules* with the release of version 2.0, and we expected some related concepts would remain in both. Additionally, both *modules* and *server* process many of the same data types. This is reflected in the types of clones we see, which primarily involve handling of the bucket brigade and translating document requests. That said, there was still relatively little cloning between the two subsystems, with a total of eight clones.

Cloning between *modules* and *test* and *support* were surprising. The *support* subsystem consists of several individual executables used for benchmarking and configuring the server. The clones we found were between the Apache Benchmark utility and the *mod_ssl* subsystem. They were simple function clones involving the setup of an ssl connection. Clones between *test* and *modules* involved converting time strings, there were only two.

Most of the clones we listed above, specifically function clones, involved generic high level concepts that should be implemented in a standard library, such as *srclib*. For most of the above redundant code, it would be sensible to factor out the commonalities to such a place. Overall, however, the cloning at the high level was mostly superficial, and not overly concerning.

Lessons Learned

Visualization using *LSEdit* is most powerful as an exploration mechanism. Using this tool, we "browsed" the dependencies within the software, noting and further investigating dependencies that seemed unreasonable or unexpected. Generally, dependencies were deemed unreasonable if they occurred between unrelated subsystems, for example those between *srclib* and other subsystems. The resulting investigation helped us locate possible points of misplaced and duplicated concepts.

Being able to query for only clones occurring between a set of entities was vital in the analysis. This feature provided easy access to the clones composing a given edge in the graph making the investigation fast and efficient.

Automatically categorizing the query results using the clone taxonomy described earlier made it very easy to determine the types of dependencies that connected the subsystems, further enabling us to understanding what was cloned and how. The taxonomy simplifies the analysis by providing structure in the form of task separation. Separating larger function clones from *clone blocks* allows us to organize the task into examining duplicated functionality, which tend to be manifested in the higher level *function to function clones* clones, and duplication due to common concerns, such as the use of a particular data structure, which tend to be manifested in *clone blocks*.

Automatic categorization also acts as rough metric for evaluating a cloning situation, particularly as a form of metric for assessing the strength of the semantic relationship between to software entities. For example, many *function clones* can indicate a stronger relationship than *clone blocks*.

Deeper Into MPM

An example where cloning across subsystems is very high within the *Apache* web server is the *server/mpm* subsystem. It is a good example of what we call *cloning "hotspots"*. A *hotspot* is an artifact (a subsystem, file, method, etc) that contains a substantially larger portion of clones than other entities near it. From Table II we see that this subsystem contains a highly disproportionate amount of cloning, with 38.8% of the overall clones in the system, but comprises only 17% of the lines of the total source code. 20% of the lines of code within this subsystem were in a clone relationship, which is 8% above the overall system average. This is another indication of high cloning activity within the software system.

This subsystem contains the implementation of several process management subsystems for the various platforms and performance requirements. Each of these subsystems is responsible for the same functionality, namely network communications and requests handling and dispatching. Each subsystem is tailored for a particular operating system or performance model, but because they are all developed for the same purpose, a high degree of cloning is expected.

In Table II we see that there is a very high degree of external cloning within the *mpm* subsystem. The view in LSEdit, shown in Figure 2(b), is one of a fully connected graph, each subsystem being

Name	Internal	External	%Clones	%Lines
httpd				
srclib	1165	22	40.8	39.8
server	1103	24	38.8	17.0
modules	513	19	18.3	40.5
support	68	7	2.6	3.1
os	9	3	0.4	1.2
test	3	6	0.3	0.6
include	0	3	0.1	1.9
server/MPM				
experimental/threadproc	11	275	6.6	1.0
experimental/leader	11	266	6.4	0.9
experimental/perchild	9	219	5.2	0.9
worker	11	281	10.1	1.2
prefork	5	222	7.8	0.6
netware	4	192	6.8	0.6
winnt	18	166	6.3	2.2
beos	5	174	6.2	0.5
mpmt_os2	1	118	4.1	0.4

Table II. Distribution in Subsystems

related to every other subsystem. It is interesting to note in Table II, there is relatively little cloning within each entity, but the external cloning is very high.

Looking at the categorized clones within this subsystem provides more detail about the cloning within *mpm*. From this we noted that 78.0% of the clones in *mpm* contribute to function clones. Also, 93% of clones in this subsystem are 2^{nd} and 3^{rd} cousin clones. The amount of duplication across subsystem boundaries is much higher than that typically seen in our case studies. However, in this case, these numbers are not surprising and are explained by the commonality of the functional requirements of the subsystems. A contributing factor to the very high number of external clone pairs is caused by cloning between the functions *ap_query*. This function is present in all of the *mpm* subsystem and composed of a *switch* block. Many clones are produced per pair of functions. While they are quite related, it seems more logical to group the many clone pairs as one clone pair. This reduces the number of clone pairs within *mpm* by 864 clone pairs. This change causes *mpm* to now contain 21% of the clone pairs. If we completely remove these functions from the analysis the percentage of cloned lines within *mpm* drops only to 19.0%, making *mpm* still a *cloning hotspot*.

Each implementation is designed to be most efficient for a particular platform, thus the main process management code is different. However, there are key interfaces with the rest of the system that must behave in a consistent manner, such as those functions related to setting the maximum number of threads or processes. Also, there are certain concepts, which are not necessarily platform specific, that will also be implemented in very similar ways, such as the functions for deciding how many new servers to spawn on the next maintenance cycle. Each of these functions exists on its own because of the platform specific variations within them, but as a whole retain similar behavior.

There are several functions that contribute to large clone groups within this subsystem, with names similar to *setserver_limit, ap_mpm_query, set_deamons_to_start, set_max_free, set_min_spare_threads* and *set_signals*. Using only clone pair relationships, these groups of similar functionality would not be found due to the variations amongst them. However, by generating equivalence classes using cloning in general as a relationship, we can recover these somewhat hidden classes of relationships from the data set.

Within the *mpm* subsystem, we find an example of a cloned subsystem. The subsystem *experimental/threadpool* is based on the source code of the *worker* subsystem. We also see that *experimental/leader* is based on *worker*. This form of cloning is an example of “good” cloning. Occasionally the goal of maintaining system stability and the goal of exploratory development may conflict. *Leader* and *threadpool* are designed to provide performance improvements on the *worker* module. In order to maintain stability as well as code clarity, a version of the module was forked and modified rather than modifying the stable code of *worker*. In this setting, this appears to be a good solution, as it avoids complicating the code with compiler directives. This type of cloning behavior was particularly interesting to us, as we had seen it before. Examples of this can be found in the Linux kernel where the *ext3* file-system was initially cloned from *ext2*.

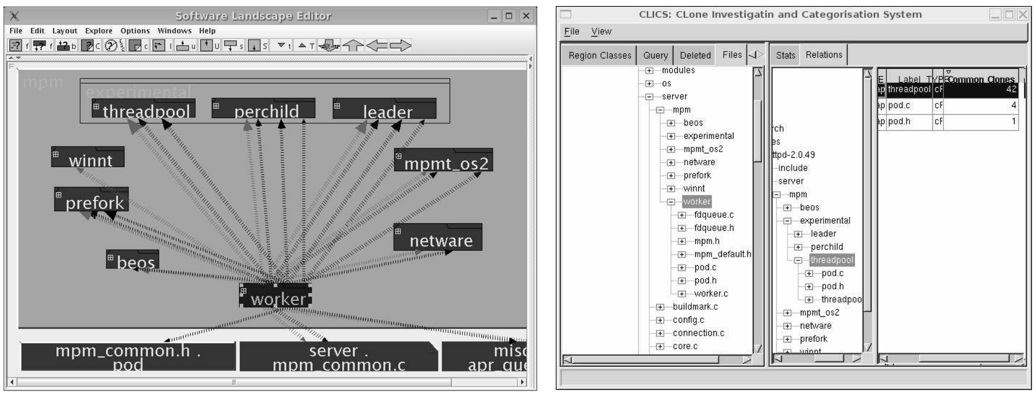
Lessons Learned

Visualization is an important tool for the identification and initial evaluation of abnormal situations. As can be seen in Figure 2(b), the clone relationships is within the *mpm* subsystem are quite complex. From the view we could see the high inter-relations amongst the components of the *mpm* subsystem. Diagrams such as Figure 2(b) can act as sirens to software maintainers as a possible indication of poor design or coding practices (which is not the case here however).

LSEdit provided an efficient way to explore the software system at a high level of abstraction and investigate overall relationships amongst subsystems. Showing cardinalities of edges on nodes provided immediate feedback as to the extent of the various relationships. In Figure 3(a) we used a forward query to see the clone relationships to the *worker* subsystem. In this diagram we chose to show cardinality using the size of the arrow head. From this figure we can see that all other subsystems have a clone relationship to worker, but *threadpool* and *leader* have a stronger relationship than the others.

The *system navigation tree* and its related subsystems tree were essential for the investigation of the *worker* module and its descendants. Using these trees is generally more suitable at lower levels of analysis, for example when locating the functions or files that are the source of most of a clone relationship between two high level subsystems. An example of this is shown in Figure 3(b). Here we can show the files and functions that are most related to *worker*. While the graph visualization is useful for analyzing the dependencies amongst subsystems, it is more difficult to show the source of these relationships when the occur deep within the containment structure.

Querying support was essential in the analysis of this system. In particular, transitive queries on the RGCs, ignoring the actual clone type, were most helpful. This type of query is able to find relationships between functions where changes in the cloned code make it difficult to detect the relationship using standard detection methods. A more loose form of a clone class is found in this way. For example, using these queries, we were able to note that the *perform_idle_server_maintenance* functions were part of one of these groups, and had some relationship to each other. However, *CCFinder* was unable to detect these relationships explicitly. Not all functions were cloned to the same degree, but most functions in the class have some relationship to each other. With such querying facilities, software maintainers will get a broader picture of the cloning in the system, enabling them to more effectively maintain redundant code.



(a) (b)

Figure 3. Viewing relationships of *worker* subsystem and navigating using *system navigation tree*.

Other Sources of Heavy Cloning

The *mpm* subsystem is not the only entity in the system where duplicate behavior through code duplication is performed. The systems *threadproc, lock, network_io* and *fileio* in *apr* are all examples of *cloning hotspots*. The subsystem *threadproc* contains 9.7% of the clones, but only 2.3% of the lines of code of the software. Similar values can be stated for the other subsystems as well. For these subsystems, the percentage of internal lines of code cloned was also very high. *threadproc* had the highest percentage of cloning with 30% of lines cloned, and *fileio* with the lowest at 16%.

These subsystems exhibit a similar distribution of clone types as *mpm*. The *apr* libraries, as mentioned above, are designed to provide a portable, consistent interface to the operating system. Common problems in C programming such as memory management, time, and strings are implemented for each of the supported platforms. In cases where the platforms are sufficiently similar, compiler directives are used to make the code portable. In cases where this can not be done, individual subsystems are made for each platform specific implementation of the feature, and in many cases common code is inevitable.

Many clones that we found in *apr* were very similar or identical. For example, 15 function clones are for memory allocation in a pool throughout Apache, often with the only change often being the parameter type. This is an example where polymorphism would be useful in solving the problem. Since procedural languages such as C do not support polymorphism, refactoring is likely to be difficult because of the many different types of structures that are passed to this function. However, documenting a list of these functions may aid in maintenance in the future if changes need to be made.

Another source of many clones, this time within the same file, was found in *apr-util/xml/expat/lib/xmlrole.c*. Part of the XML parser for Apache, this file contains 244 *function clones* amongst 43 out of 53 functions. All of these functions are 15–20 lines long, containing primarily a switch statement for handling a token. Because all of these functions are structured in the same way, and in many cases use similar or the same constants in their *switch* statements, we do not classify these as false positives.

Lessons Learned

Again, we found that transitive queries were very useful in this portion of the study. We also noticed the need for an additional feature: automatic extraction of a template of the cloned code would be beneficial as both an analysis and a refactoring tool. This will be the subject of future work.

Based on the observations stated above that *cloning hotspots* tend to have both disproportionate amounts of cloning, and also higher coverage of lines by clones, we believe it is possible to automatically detect and present these *hotspots* to the user. This will be an extension of the tool left for future work.

Equivalence Classes of RGCs

One drawback to our current methods of clone navigation is the structuring around clone pairs only. During the course of our study, transitive queries were often necessary to gain an understanding about the cloning in the subsystem, as opposed to just between two software entities. This prompted us to create a way of navigating clones using the groups generated by these queries.

The groups are generated using the most general transitive query our system supports. In this query, any two regions that share code are considered to have a clone relation. Using this general relation, we generate the equivalence class, which we call a clone class, for all regions extracted from the source code. Clone classes are a common way of grouping clones, and were first proposed by Mayrand et. al [26]. Sorted by size, we can then browse these classes. We were able to make several qualitative observations in our case study. One might expect that this grouping technique would be too general and produce large clone classes containing unrelated regions of code. In fact, this turns out to be a rare occurrence in the case study. In general, the clone classes are composed of highly related regions. However, in the case of the largest clone classes, they are not strict clones, but rather are functions dealing with very similar concepts, such as with setting the maximum number of threads or children the server can use.

We also noticed that clone classes tend to reside within the same subsystem, or parent subsystem, further evidence that Apache is well organized. There were very few examples where clone classes crossed high level subsystem boundaries. Table III shows a summary of the sizes of clone classes we extracted. We have defined the size of a clone class to be the number of unique clone regions it contains. In this table, there is a row for each size of clone class that we saw. Each row shows the average maximum distance of the clone classes of that size. Maximum distance for each clone class is taken from the RGC with the highest cousin relationship as described by the clone taxonomy. For example, if the maximum distance for a clone class is a *2nd Cousin* clone, then the maximum distance will be 2. The average number of RGCs and the total number of clone classes of this size is also shown.

From Table III, we see that clone classes most often stay within the same directory. The table shows that even for clone classes as large as 26 regions, the maximum distance is a *2nd cousin* clone. We can also see from this table that the average size of a clone class is 2.51 regions, showing us that in general functions do not get repeatedly cloned.

Summary

Upon initial inspection, *Apache* appears to deviate from previous findings that cloning tends to occur within subsystems. However, closer inspection reveals that this observation is mostly true for Apache. While cloning was most often found to occur between two distinct subsystems, most subsystems sharing code were contained within the same higher level subsystem. In fact, as described in our case

# Regions	Average Max Distance	Average # RGCs	# Clusters	# RGCs
1	0.00	1.00	108	108
2	0.86	1.10	216	239
3	1.18	2.98	61	182
4	1.33	5.46	15	82
5	1.12	7.75	8	62
6	1.71	12.14	7	85
7	0.60	11.40	5	57
9	1.00	31.33	3	94
10	0.00	18.00	1	18
11	0.00	32.00	1	32
12	2.00	68.00	1	68
13	1.50	45.50	2	91
15	1.50	55.50	2	111
21	0.00	210.00	1	210
23	4.00	45.00	1	45
26	2.00	112.00	1	112

Table III. Summary of Clone Classes

study, cloning across the highest level subsystems was quite rare. Described in the taxonomy as *4th* and *5th Cousin Clones*, these clones only account for 1% of the total overall cloning.

The *Apache* case study has raised some interesting questions about cloning in multi-platform software systems. In this study, we found that platform–specific code often had a high degree of cloning. It appears that such cloning is a reasonable design strategy, in terms of flexibility and design of the software system. Activities like this provide a way to "bootstrap" the porting of platform specific code, without requiring major changes to the design of the overall system. This can be an advantage in the initial stages of development when appropriate abstraction levels and degrees of commonality between subsystems are unclear. In later stages of the program development cycle, this can still be an appropriate method of duplicating behavior in a software system. In cases of experimental additions to the system, such as *mpm/experimental*, it is reasonable to clone code because prototypes or exploratory projects should not infect the currently stable and maintainable code.

Related Work

Visualization of clones is commonly done using scatter-plots to present matched lines of code [5, 11, 27, 28] These scatter-plots provide the ability to select and view clones, as well as zoom in on regions of the plot. In practise, we have found scatter-plots do not scale well with medium to large software systems, the points become so small that it is difficult to pick out all but the most severe cloning. Additionally, scatter-plots do not easily lend themselves well to providing the context of cloning from an architectural perspective.

Gemini [28] and Aries [14] are two tools that use CCFinder as their core clone detection mechanism. In addition to scatter-plots, Gemini also provides visualization through metrics graphs and file similarity tables. It allows users to browse clones either pair-by-pair, or using clone classes. Aries is a refactoring support environment for duplicated code. Aries supports refactoring using metrics–based querying. Users can query for clones matching a variety of metrics and thresholds. An important feature that of Aries is that it can recommend a refactoring method to use based on the metrics of the code clones. Another difference in the tools proposed is that while Aries provides the capability to refine the displayed clones using queries, these tools do not support data set refinement or views mapping clones to concrete architecture.

In [16], Johnson used Hass diagrams to visualize cloning relationships. In [17], he proposed the use of hyper-linked documents to navigate cloning relationships. Reiger et. al. describes five polymetric views with the focus of showing what parts of the system are connected via code cloning and what parts are cloned the most [27]. These views have been designed to educate the user about the cloning in a software system at different levels of abstraction, providing progressively more information about the cloning in the software. Using metrics, architectural graph representations and the *System Navigation Tree* we also provide the first four views. Our work differs from the above works in that we aim to provide the criteria required to make a complete clone comprehension tool. Providing high level views and navigation through visualization is one part of the overall system. We also require filtering facilities, metrics reporting, querying facilities.

Clone detection case studies on the Linux kernel have been reported in [4, 10, 12]. In [10], Casazza et al. use metrics based clone detection to detect cloned functions within the Linux kernel. 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. [4] did a similar study, evaluating the evolution of code cloning in the Linux, concluding that the structure of the Linux kernel did not appear to be degrading due to code cloning activities. In [13] a preliminary investigation of cloning among Linux SCSI drivers was performed. Other recent studies have shown that cloning in software tends to occur between files that are close within the system [18, 19, 20]. From our case study, we see that the *Apache* development team did tend to clone when adding code related to a specific platform.

There is a wide variety of clone detection techniques that have been developed. These methods range from string comparison, metrics comparison, and program graph comparison strategies [7, 5, 8, 11, 15, 18, 22, 24, 25, 26]. Clone classification schemas have been previously suggested, usually based on the degree of similarity of segments of code and also the type of differences [6, 26]. These classifications are limited to function clones only. In [6], 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 [6] are used by Balazinska et al. in [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 [6] they break this down into 18 categories. In the future, our taxonomy will also include a finer categorization of function clones.

Conclusions

Cloning in software systems is an important maintenance challenge. It requires tool support to make analysis and management tractable. The purpose of this study was to propose a set of criteria for tools designed to aid in the understanding of clones in a software system. We described the types of features that could be used to meet these criteria and then demonstrate the use of prototype of a tool designed to meet those criteria. Through a case study, we show the value of many of these features at different levels of the analysis. We also identified other features of a clone analysis system that may be useful.

During our study, we found that cloning very often occurred between related subsystems, and tended not to cross major subsystem boundaries. Architectural views are useful in spotting usual cloning that does not hold to this observation. These views are also useful in locating similar subsystems and justifying the cloning between them.

In this paper we have provided a description of a taxonomy of clones that takes into account the type of code the clones occur in, in addition to location within the system and the degree of similarity. This taxonomy was useful as a learning mechanism and a method of extracting useful metrics about the cloning within the software. It was also useful as a exploratory device. When used to classify query results the taxonomy simplified the problem of inspected the clones.

ACKNOWLEDGEMENTS

We would like to thank Inoue et. al. for the use of CCFinder in this study.

An earlier version of this paper appeared in IEEE International Conference on Software Maintenance [21].

REFERENCES

1. *The Apache HTTP Server Project,* `"http://httpd.apache.org/"` May 2005.
2. *Netcraft: Web Server Survey Archives,* `"http://news.netcraft.com/archives/web_server_survey.html"`, May 2005.
3. *A Taxonomy of Clones In Software* `"http://swag.uwaterloo.ca/~cjkapser/CLICS/taxonomy"`, May 2005.
4. Antoniol G, Villano U, Merlo E, Di Penta M. Analyzing cloning evolution in the Linux kernel. *Information & Software Technology* 2002, **44**(13):755–765.
5. Baker B. S. On finding duplication and near-duplication in large software systems. *Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95)*, 1995. IEEE Computer Society: Toronoto, Ontario, Canada, 1995; 86–95.
6. Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K. Measuring clone based reengineering opportunities. *Proceedings of the Sixth International Software Metrics Symposium*, Nov 1999. IEEE Computer Society: Boca Raton, Florida, USA, 1999; 292–303.
7. Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K. Advanced clone analysis to support object-oriented system refactoring. *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, 2000. IEEE Computer Society: Washington, DC, USA, 2000; 98–107.
8. Baxter I,Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. *Proceedings of the International Conference on Software Maintenance (ICSM '98)*, 1998. IEEE Computer Society: Brisbane, Queensland, Australia, 1998; 368–377.
9. Bowman I, Holt R, Brewster N. Linux as a case study: its extracted software architecture. *Proceedings of the 21st international conference on Software engineering (ICSE '99)*, May 1999. IEEE Computer Society Press: Los Angeles, CA, US, 1999; 555–563.
10. Casazza G, Antoniol G, Villano U, Merlo E, Di Penta M. Identifying clones in the Linux kernel. *First IEEE International Workshop on Source Code Analysis and Manipulation*, Nov 2001. IEEE Computer Society Press: Florence, Italy, 2001; 92–100.
11. Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, Aug 1999. IEEE Computer Society: Oxford, England, UK, 1999; 109–118.
12. Godfrey M, Svetinovic D, Tu Q. Evolution, growth, and cloning in Linux: A case study. *A presentation at the 2000 CASCON workshop on 'Detecting duplicated and near duplicated structures in large software systems: Methods and applications', on November 16, 2000, chaired by Ettore Merlo; available at* `http://plg.uwaterloo.ca/~migod/papers /cascon00-linuxcloning.pdf`
13. Godfrey M, Tu Q. Evolution in open source software: A case study. *Proceedings of the International Conference on Software Maintenance (ICSM '00)*, Oct 2000. IEEE Computer Society: San Jose, California, USA, 2000; 131–142.
14. Higo Y, Kamiya T, Kusumoto S, Inoue K. ARIES: Refactoring support environment based on code clone analysis. *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications (SEA 2004)*, Nov 2004. ACTA Press: Cambridge, MA, USA, 2004; 222–229.
15. Johnson J H. Substring matching for clone detection and change tracking. *Proceedings of the International Conference on Software Maintanence*, Sept. 1994. IEEE Computer Society: Victoria, BC, Canada, 1994; 120–126.
16. Johnson J H. Visualizing textual redundancy in legacy source. *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research (CASCON '94)*, Oct 1994. IBM Press: Toronto, Ontario, Cananda, 1994; 9–18.

17. Johnson J H. Navigating the textual redundancy web in legacy source. *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research (CASCON '96)*, Oct 1996. IBM Press: Toronto, Ontario, Cananda, 1996; 7–16.
18. Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 2002; **28**(7):654–670.
19. Kapser C, Godfrey M. Toward a taxonomy of clones in source code: A case study. *Evolution of Large Scale Industrial Software Architectures (ELISA '03)*, Sept 2003. Amsterdam, The Netherlands, 2003; 67–78.
20. Kapser C, Godfrey M. Aiding comprehension of cloning through categorization. *Proceedings of 7th International Workshop on Principles of Software Evolution (IWPSE '04)*, Sept 2004. IEEE Computer Society: Kyoto, Japan, 2004; 85–94.
21. Kapser C, Godfrey M. Improved tool support for the investigation of duplication in software. *Proceedings of the International Conference on Software Maintenance (ICSM '05)*, Sept 2005. IEEE Computer Society: Budapest, Hungary, 2005; 305–314.
22. Komondoor R, Horwitz S. Using slicing to identify duplication in source code. *Proceedings of the 8th International Symposium on Static Analysis (SAS '01)*, July 2001. Springer-Verlag: Paris, France, 2001; 40–56.
23. Kontogiannis K. Evaluation experiments on the detection of programming patterns using software metrics. *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE '97)*, Oct 1997. IEEE Computer Society Press: Amsterdam, The Netherlands, 1997; 44–55.
24. Kontogiannis K,DeMori R, Merlo E, Galler M, Bernstein M. Pattern matching for clone and concept detection. *Automated Software Engineering* 1996; **3**(1/2): 77–108.
25. Krinke J. Identifying similar code with program dependence graphs. *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE '01)*, Oct 2001. IEEE Computer Society: Suttgart, Germany, 2001; 301–309.
26. Mayrand J, Leblanc C, Merlo E. Experiment on the automatic detection of function clones in a software system using metrics. *Proceedings of the International Conference on Software Maintenance (ICSM '96)* Nov 1996. IEEE Computer Society: Monterey, CA, USA, 1996; 244–253.
27. Rieger M, Ducasse S, Lanza M. Insights into System-Wide Code Duplication. *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)* Nov 2004. IEEE Computer Society: Delft, The Netherlands, 2004; 100–109.
28. Ueda Y, Kamiya T, Kusumoto S, Inoue K. Gemini: Maintenance support environment based on code clone analysis. *Proceedings of the Eighth IEEE Symposium on Software Metrics*, June 2002. IEEE Computer Society: Ottawa, Canada, 2002; 67–76.

AUTHORS' BIOGRAPHIES

Cory J. Kapser graduated from the University of Alberta with a B.Sc. in Computer Science in 2002. He is currently pursuing a Ph.D. at the David R. Cheriton Computer Science, University of Waterloo under the supervision of Dr. Michael Godfrey. Currently he is interested in analysis and comprehension of large software systems.

Michael W. Godfrey is an assistant professor in the David R. Cheriton School of Computer Science at the University of Waterloo. He is the associate chairholder of the Industrial Research Chair in Telecommunications Software Engineering sponsored by Nortel Networks, the National Science and Engineering Research Council (NSERC), and the University of Waterloo. He holds a Ph.D. in Computer Science from the University of Toronto (1997) and has also been a faculty member at Cornell University. His research interests include software architecture extraction and modelling, reverse engineering, software evolution, and program comprehension.