International Journal of Security and Its Applications
Vol. 9, No. 6 (2015), pp. 167-182
http://dx.doi.org/10.14257/ijsia.2015.9.6.17

Extract Function Clone Genealogies across Multiple Versions

Tu Ying[1], Zhang Li-ping[2], Wang Chun-Hui[3] and Liu Dong-sheng[4*]

[1,2,3,4]*Department of Computer & Information Engineering*
Inner Mongolia Normal University
[1]*ty_sky0908@163.com*, [2]*cieczlp@imnu.edu.cn*, [3]*ciecwch@imnu.edu.cn*
[4]*nmlds@126.com*

Abstract

Software systems often contain plenty of code clones, which bring significant impact on software development and maintenance. Tracking clones in the evolution process is essential to analyzing clones, since we cannot understand clone phenomenon well just rely on the clone detection results of single version. We developed a function clone genealogy extractor, cGen, which can track clones across multiple versions to extract type-1 and type-2 function clone genealogies. By using cGen, we examine nine open source C projects and analyze their evolution. Our study shows that cGen can efficiently extract clone genealogies from multiple versions of a project, and provide support for clone evolution analysis.

Keywords: *code clone; clone genealogy; clone evolution; clone management*

1. Introduction

Code clones are common in all kinds of software systems, since programmers often copy and paste code fragments during software development [1]. Such a code reuse mechanism is easy and cheap, but it makes software maintenance more complicated [2]. On the on hand, reusing some well-tested code can reduce the potential of the risk of writing new code, and save the development cost. On the other hand, if the original code contains a defect, that could cause propagation of the defect. Furthermore, clones can increase the risk of introducing defects by inconsistent changes to parts of the code that are meant to evolve identically [3]. For its significant impact on software quality, analysis of code clones has become a very active research branch of software analysis [4].

An abundance of studies have been conducted on clone detection, refactoring, and clone evolution, *etc*. In recent years, there seems to be a general consensus among researchers that comprehensive management of clones is desirable rather than aggressively refactor entire clone detection results. Clones are not always harmful, manage clones that have no negative effects creates only additional effort [5]. Further understanding of clone features is crucial for comprehensive management of clones. Since clone detection results of single version cannot provide us sufficiently information, tracking clones in evolving system can help us obtain more useful insights [6]. To investigate clone evolution, we choose to use clone genealogy model, which was first presented by Kim *et al.*, and has been applied to investigate clone evolution by some other researchers. The genealogy of code clones describes how groups of code clones change over multiple versions of a program [7]. From the perspective of evolution, it is able to longitudinally analyze patterns and characteristics clone exhibited over the lifetime of software, which makes a valuable contribution to manage clones more efficiently.

In this paper, we study genealogy extraction of function clones. Two functions are considered as clones when the body of the functions consists of codes that are similar

* Corresponding Author

ISSN: 1738-9976 IJSIA

enough [8]. We select the function granularity because our previous studies were conducted on systems written in C language, and each C source file can be composed of one or more functions. But our extract method is flexible and easy to expand to other programming languages like C++ or Java, *etc*. In order to analyze clone evolution, we identified several evolution patterns for code clones. We partly use the evolution patterns described by Kim et al. but more than that. Existing patterns are either defined for clone fragments or for clone groups. Still, many of these patterns are similar or strong related. We are motivated by the open questions presented by Nils Göde in his PhD thesis [9]. He summarized that basic clone evolution patterns that have been described in the literature and argued that it would be beneficial to represent only low-level patterns for clone fragments. And further in-depth analyses that investigate the change patterns in the evolution of individual clone fragments can suggest techniques for optimizing clone management including refactoring and removal [10]. On this basis, we proposed four low-level patterns for clone fragments, which allow taking steps to derive high-level patterns for clone groups. We implemented a clone genealogy extractor for function clones, named cGen. Using cGen, we extracted clone genealogies of six open source C systems, each of them has multiple release versions. The experiment results indicate that cGen can efficiently extract clone genealogies form a series of versions of a project, and serve as a fundamental tool for analysis of function clones in C projects.

2. Related Work

The literature reports several studies concerning clone genealogy, and tracking clones in evolving system.

2.1. Clone Genealogy

The first study of code clone genealogy was presented by Kim *et al.*, [7]. They proposed clone genealogy as a formal definition of clone evolution, and built a clone genealogy extractor. This tool extracts all versions of a project from its source code repository (CVS). Using Kenyon's front-end and clone detection tool CCFinder [11], they tracked the evolution of code clones in two Java programs and analyzed how clone groups evolve with software evolution. They concluded that many clones may not be worthwhile to extensive refactoring, and kind of clones are not easily to refactor due to programming language limitations. Then clone genealogy became one of the basic methods for research clone evolution. The extraction of clone genealogies from a series of versions of a program has been identified as the fundamental task to study the evolution of individual clone groups.

Saha *et al.*, [12] extended the study by Kim, they conducted an in-depth study on the evaluation of clone genealogies in 17 open source systems cover four different programming languages. Their findings further confirmed Kim's insights, clones may not always be harmful in software maintenance, and we should possibly focus on tracking and managing clones in evolving system instead of aggressively refactoring them.

2.2. Tracking Clones in Evolving System

An essential prerequisite for extracting clone genealogy is being able to build clone mappings, that is, tracking clone between two adjacent versions. Accuracy of the mapping plays a significant role in clone evolution researches, directly affects the research value of the subject [13].

Duala-Ekoko *et al.*, [14] proposed a method to track clones in evolving system, relied on the concept of clone region descriptors (CRDs), which described clone regions independent from the exact text of clone regions or their location in a file.

Nils göde [15] presented a method to map clones between consecutive program versions during the clone detection process. Clones are mapped according to the changes made to the source files of the program between versions.

Bakota [16] presented an approach for mapping code clones from one particular version of the software to another one, based on a similarity distance function. The similarity distance function considering the lexical structure of clone fragments, combined with information obtained from abstract syntax tree.

Saha *et al.*, implemented a near-miss clone genealogy extractor, named gCad [17]. For two given consecutive versions, gCad first extracted functions from both versions. Then do the clone mapping on the basis of function mapping to obtain fast computation.

Ci Meng [18] improved the CRDs model proposed by Duala-Ekoko *et al.*, and used it to describe code clones. They proposed a CRD-based clone group mapping algorithm to map clone groups between neighboring versions, and finally, constructed clone genealogy based on binary relation combination.

In the case of tracking clones across different versions of a software system, existing methods either rely on commits in CVS or SVN, or need higher computation cost. In this paper, we present a token-based clone mapping method independent of information from version control systems. It can efficiently construct the mapping relationship between two adjacent versions, and then as basis of extracting genealogies from multiple versions of a system.

The most similar work with ours was Saha *et al.*, [12, 17], but with below two main differences, as below:

(1) They use TXL [19] to extract all the function signatures, first map functions between two versions, then do the clone fragment mapping on the basis of function mapping. So the computation speed is very fast. However, if a function was renamed, or moved to other files, it would influence the accuracy of clone mapping. Instead, we map clone groups between two versions based on their token representation firstly, then extract location information of each fragment for clone fragment mapping. In this way, it is not easy to lose the relationship between clone fragments, and take the advantage of low computation at the same time, since the number of clone groups is much less than the number of clone fragments.

(2) They focused on the category of clone genealogies, classified them into four types, including alive genealogy, dead genealogy, syntactically similar genealogy and consistently changed genealogy. While our study focused more on function clone genealogies and analyzed various evolution patterns described for clone groups. Particularly, we analyzed clone lineages specific to current version.

3. Study Approach

Our approach for extracting clone genealogies mainly consists of the following three steps:

(1) Clone mapping phase

(2) Clone evolution patterns identification phase

(3) Clone genealogies extraction phase

Clone fragment is the basic entity of clone evolution analysis. In this paper, each clone fragment is exactly a function in C source code. Because our study conducted on type-1 and type-2 function clones in C projects. A clone group comprises two or more clone fragments that are clones of each other. For convenience, in the following, clone fragment can be abbreviated to CF, clone group can be abbreviated to CG. And we assume that *OG* is a clone group in V_i, and *NG* is the corresponding clone group in V_{i+1}. Figure 1 shows the general roadmap of our study.

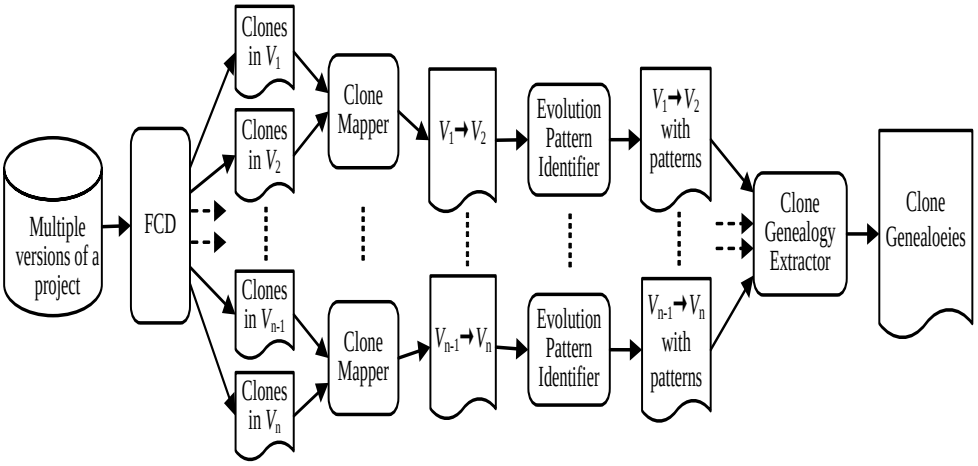

Figure 1. General Roadmap of the Study

3.1. Clone Mapping

Clone genealogy extraction needs to track clones across multiple versions, that is, map a certain clone from a version to the corresponding clone in the next version. Mapping clones between two neighboring versions of a system is the foundation of tracking clones, as well as a fundamental question in extracting clone genealogies. So the accuracy of clone mapping is the key to ensure the correct extraction of clone genealogies. In this paper, we present a token-based clone mapping method, which maps clones at fragment level on the basis of preliminary mapping results at group level. Using a combination of properties of clones, such as text, lexical structure, contained file name and function name, first mapping clone groups before mapping each clone fragment. Figure 2 shows the clone mapping process between two neighboring versions, including the acquisition of available clone detection results, the establishment of clone group mapping and clone fragment mapping, as described below.

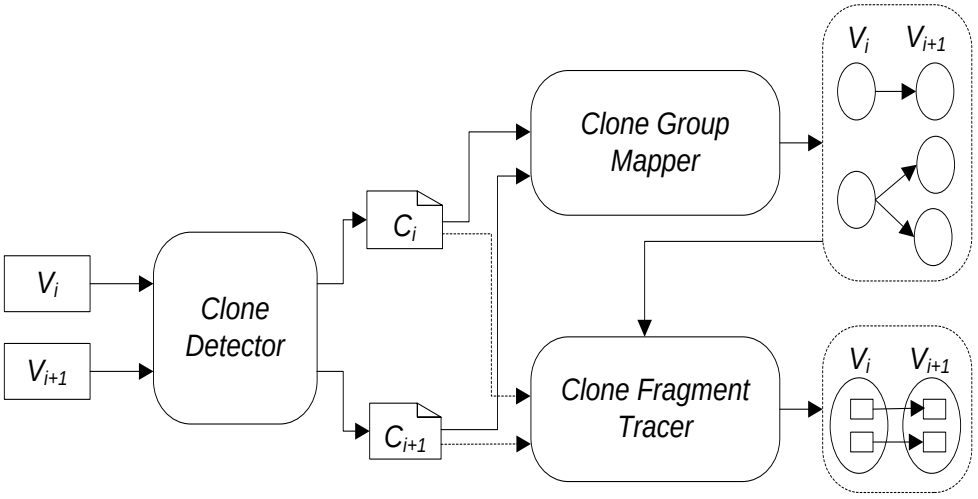

Figure 2. Clone Mapping Process between Adjacent Versions

3.1.1. Obtain Clone Detection Information: Clone detection results of each version of a system are the prerequisites for clone evolution research. We use our clone detector FCD [20] (Function Clone Detector) to obtain clone detection results. FCD implemented based on optimized suffix array, it can efficiently detect type-1 and type-2 function clones in C systems, and with good recall and precision. Since FCD only detect a single version at a

time, we write a shell script to automatically detect all given versions of a system, as shown in Figure 3. Putting all versions of a system in the same directory, then one can use the shell script to automate the multi-version detection process. By pre-processing the clone detection results, some valuable information can be obtained for the following research, including detected clone groups and their token representation, and the location of each clone fragment, *etc*.

```
#!/bin/bash
read UserDir
for version in $UserDir/*
do
  if [ -d "$version" ] ; then
  {
    echo Version $number
    echo $version
    ./FCD $version
    ((number+=1))
  }
  fi
done
```

Figure 3. A Shell Script for Automatic Multi-Version Detection

3.1.2. Build Clone Group Mapping: After obtained all information needed from the clone detection phase, we build the clone group mapping next. As FCD, the tool we used for clone detection, is a token-based clone detector, all clone fragments in the same group have identical token sequence. For type-1 and type-2 clones, so token is considered a good representation of clone group. To trace a clone group across versions, we define a *TokenSimilarity* function that measures the token similarity between two tokens T_1 and T_2, that is, the similarity between two clone groups CG_1 and CG_2. Here T_1 and T_2 are token representation of CG_1 and CG_2 respectively. We apply the longest common subsequence (LCS) algorithm to calculate the common ordered tokens between T_1 and T_2.

$$TokenSimilarity(T_1, T_2) = \left\{ \frac{LCS(T_1, T_2)}{len(T_1)} + \frac{LCS(T_1, T_2)}{len(T_2)} \right\} / 2 \quad (1)$$

Equation (1) describes the *TokenSimilarity* function. Where $LCS(T_1,T_2)$ denotes the length of longest common subsequence of T_1 and T_2. Given both token of two clone groups, it calculates a value between 0 and 1 to reveal how similar this two clone groups are in terms of lexicon.

In this paper, we choose LCS dynamic programming algorithm, time complexity is O(m*n). And utilize roll array to optimize data storage space, as shown in Figure 4, reduce space complexity to O(n).

When a clone group changed during software evolution, it may have some difference with the clone group in the next version with regard to its tokens representation. We set a token similarity threshold *t*, considered all clone group pairs with *TokenSimilarity* value greater than *t* as mapping candidate, where *t* is a constant between 0 and 1. If a clone group may have clone mapping relationship with more than one clone group in the next version, ambiguity exists. In this case, we also use the space-optimized LCS algorithm to calculate source code similarity for all possible candidates. Select the candidate with maximum source code similarity as final clone group mapping result.

Function - calculate the length of common ordered tokens between T_1 and T_2

```
int dp[2][MAX_LENGTH];
int LCS(char* T1, char* T2, int m, int n)
{
    int i,j;
    memset(dp,0,sizeof(dp));
    for(i=1;i<=m;i++)
    {
        for(j=1;j<=n;j++)
        {
            if(T1[i-1]==T2[j-1])
                dp[i%2][j]=dp[(i-1)%2][j-1]+1;
            else
                dp[i%2][j]=max(dp[i%2][j-1],dp[(i-1)%2][j]);
        }
    }
    return dp[m%2][n];
}
```

Figure 4. Space-optimized LCS Algorithm Implement in C

3.1.2. Trace Clone Fragment: To study fine-grained characteristics of clone evolution, we utilize location features of each clone fragment to trace it across multiple versions. We select some attributes integrated represent location of a clone fragment, comprising: (a) *filename*, (b) *functionName*, (c) *startLine* and (d) *endLine* (e) *size*. For the detail description, see the table below.

Table 1. Location Attributes used to Trace Clone Fragment

Attributes	Type	Description
fileName	string	Name of the file containing the clone fragment
functionName	string	Name of the function containing the clone fragment
startLine	int	Start line number of the clone fragment in the contained file
endLine	int	End line number of the clone fragment in the contained file
size	int	Lines of clone fragment code

While the clone detection phase can provide us the complete directory clone fragment resides, the start and end line number of the clone fragment in the file, the *startLine* and *endLine* attributes can be obtained directly. Other two attributes need to be extracted further. Since we have the complete directory, split it with '\' character, the last word should be the *fileName* attribute. And because our study was conducted on function clones, each clone fragment was exactly a function. Using existing location information together we can locate source code of the clone fragment in the file, and then extracted name of the function, get the *functionName* attribute.

For two given clone groups, the old group (*OG*) in V_i and the new group (*NG*) in V_{i+1}, match the location attributes for all possible clone fragment pairs $<CF_1,CF_2>$, where CF_1 was a member of *OG* and CF_2 was a member of *NG*. If a clone fragment pair matches successfully, it means that the latter one has evolved from the first one. As we have already built the clone group mapping, the matching work only needs to be performed on

two clone groups with clone mapping relationship, which greatly reduced the number of matching. But it is still hard to do the matching work. Not just because the location of a clone fragment may change, but also its contained file name, function name, *etc*. A clone fragment may be moved to different file thus the *fileName* attribute may be changed. And the contained function may be renamed, but even after modification, the name of function usually much relevant to its original one. Because it normally with functional relevance when naming a function.

We use the following algorithm in case of the changes happened to determine whether two *functionName* attributes match. Premise of calling this algorithm is two clone fragments with non-identical *functionName* attributes. The algorithm calling C library function *strstr*, it will find a substring within a string, and returns a pointer to the beginning of the substring or NULL if not found.

```
boolean isMatched(CF1.functionName, CF2.functionName) {
    return ((strstr(CF1.functionName, CF2.functionName) != NULL)
            OR   (strstr(CF2.functionName,   CF1.functionName)   !=
NULL))
}
```

In order to get higher matching accuracy, we define a *SizeSimilarity* function (see equation (2)) to measure the size similarity of two clone fragments, filter those pairs with *SizeSimilarity* score less than 0.3 because of the significant difference.

$$SizeSimilarity(CF_1, CF_2) = \frac{\min(CF_1.size, CF_2.size)}{\max(CF_1.size, CF_2.size)} \quad (2)$$

For all possible matching clone fragment pairs $<CF_1, CF_2>$, since the premise is that their contained clone groups have mapping relationship, so their token representations are pretty similar too. We applied following matching rules organized from highest to lowest priority:

(1) If $CF_1.fileName = CF_2.fileName$ and $CF_1.functionName = CF_2.functionName$, match $<CF_1,CF_2>$.
(2) If $CF_1.fileName = CF_2.fileName$ and $isMatched(CF_1.functionName,CF_2.functionName) = true$ and $SizeSimilarity(CF_1,CF_2) > 0.3$, match $<CF_1,CF_2>$.
(3) If $CF_1.fileName \neq CF_2.fileName$ and $CF_1.functionName = CF_2.functionName$ and $SizeSimilarity(CF_1,CF_2) > 0.3$, match $<CF_1,CF_2>$.
(4) If $CF_1.fileName \neq CF_2.fileName$ and $isMatched(CF_1.functionName,CF_2.functionName) = true$ and $SizeSimilarity(CF_1,CF_2) > 0.3$, match $<CF_1,CF_2>$.

3.2. Clone Evolution Patterns Identification

Clone evolution patterns can be used to study the clone phenomenon, help people understand the patterns and features clone exhibited during the software evolution. Currently, researchers have defined some clone evolution patterns, some defined for clone fragments whereas others defined for clone groups. Still, these patterns are related. It would be beneficial to represent only low-level patterns for clone fragments. In that case, we proposed four low-level patterns for clone fragments, which allow take steps to deriving high-level patterns for clone groups.

3.2.1. Clone Fragment Evolution Patterns: After complete clone mapping relationship between any two neighboring versions have been established, we identified following four low-level evolution patterns for each clone fragment based on the mapping result.

- *Appear*: clone fragment *f* was new created in *NG*, as shown in Figure 5(a).
- *Disappear*: clone fragment *f* was changed or removed from *OG*, does not belong to *NG* any more, Figure 5(b) shows such a situation.
- *Static*: clone fragment *f* remains unchanged from *OG* to *NG*, the situation is illustrated in Figure 5(c).
- *Change*: clone fragment *f* changed from *OG* to *NG*, such as statements being added, *etc*. An example is given in Figure 5(d).

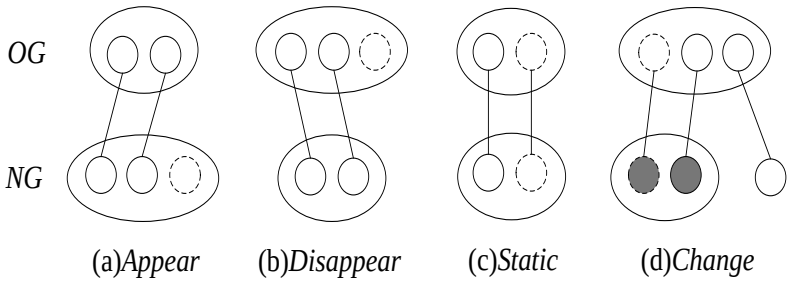

Figure 5. Clone Fragment Evolution Patterns

In Figure 5, each ellipse represents a clone group, each circle inside represents a clone fragment which is a member of the group. Occurrences of clone fragment *f* are shown as dotted circles. Different gray level means the clone fragment has changed to some extent, as well as figures following.

We set a flag array (Boolean variables) to mark whether a pattern occurred or not. If a certain pattern occurred, set the corresponding flag to True, otherwise to False.

3.2.2. Clone Group Evolution Patterns: Low-level evolution patterns were defined for clone fragments. Hence they can not reveal how fragments change in a clone group with respect to other fragments in the same group. We adopt following five clone group evolution patterns first presented by Kim *et al.*, and have been widely used to study clone evolution.

- *Same*: all clone fragments in *NG* did not change from *OG*, as shown in Figure 6(a).
- *Add*: at least one clone fragment in *NG* is newly added. For example, programmers copied a clone fragment in *OG* again when upgrading to a later version, as shown in figure 6(b).
- *Subtract*: at least one clone fragment in *OG* has changed or removed, thus it does not appear in *NG* as illustrated in Figure 6(c).
- *Consistent Change*: all clone fragments in *OG* have changed consistently, thus they belong to *NG* together due to maintaining similarity, as shown in Figure 6(d).
- *Inconsistent Change*: at least one clone fragment in *OG* changed inconsistently, thus it does not belong to *NG* anymore. For example, one clone fragment in *OG* remains unchanged while others changed consistently, as shown in Figure 6(e).

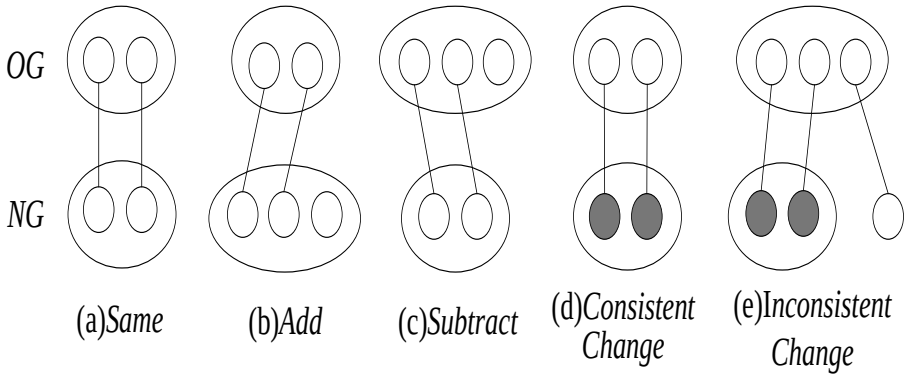

Figure 6. Clone Group Evolution Patterns

These five patterns described all possible changes to a clone group. Relationship among these patterns illustrated in Figure 7. To clarify, during the clone evolution, different kinds of evolution patterns may overlap. That means a change to a clone group may represent more than one evolution pattern. For example, *Inconsistent Change* pattern always imply *Subtract* pattern in the meantime.

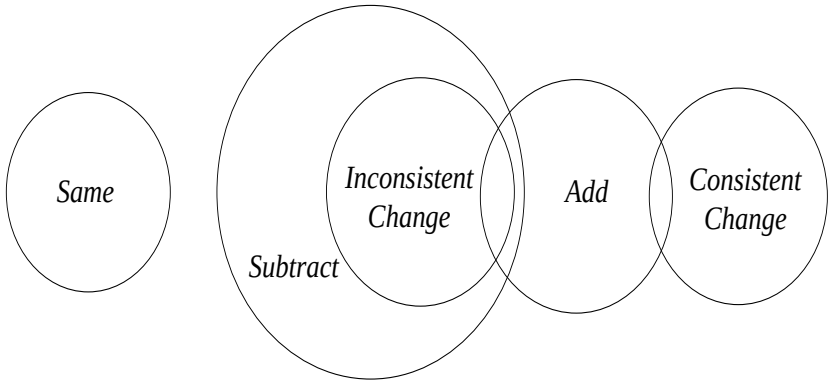

Figure 7. The Relationship among Clone Group Evolution Patterns

In this paper, we present a multi-iteration method to efficiently identified high-level evolution patterns defined for clone groups. The multi-iteration identification method consisted of three iterations, goes step by step, from simple to complicated, mainly divided based on the relationship among these patterns. Some patterns are independent to each other, such as *Same* pattern and *Consistent Change* pattern, if *Same* pattern has added to a clone group, *Consistent Change* cannot happen. So we filter the independent patterns to make the process clear and get rid of extra efforts. Every step we either do the identify activity or the filter activity.

In the first iteration, we identify the *Add*, *Subtract* and *Same* patterns. If some clone fragment in *NG* has marked *Appear*, the *Add* pattern identified. And if some clone fragment in *OG* has marked *Disappear*, the *Subtract* pattern identified. Then filter clone groups with *Add* and *Subtract* patterns, identify the remaining clone groups. If every clone fragment in *NG* has marked *Static*, the *Same* pattern identified.

In the second iteration, we identify the *Consistent Change* pattern. Due to it independent of *Same* and *Subtract* patterns. First filter clone groups have been identified as *Same* or *Subtract* pattern in the first iteration. Then identify *Consistent Change* pattern, if all clone fragments in *OG* has marked *Change*, in addition, they all belong to *NG* together, it means that the similarity does not lose between them, the *Consistent Change* pattern identified.

In the third iteration, we identify the *Inconsistent Change* pattern. A little special it is a subset of *Subtract* pattern. So we filter all clone groups except those identified *Subtract* pattern. On this basis, if some clone fragment in *OG* has marked *Changed*, it means that at least one clone fragment in *OG* has changed different from other fragments in the same group. Otherwise they will belong to *NG* together, which contradict with condition that the *Subtract* pattern occurred. So the *Inconsistent Change* pattern identified.

3.3. Clone Genealogies Extraction

At this point, we have all the detailed mapping results and evolution patterns information between any two adjacent versions. A **Clone Lineage** is a directed acyclic graph that describes the evolution history of a clone group. A **Clone Genealogy** is a set of clone lineages that have originated from the same clone group. According to the time sequence of software evolvement, cGen combines all of the results of each version pair by matching clone group ids to form clone lineages. After that, cGen extract clone genealogies by recognize clone lineages with the same origin. Figure 8 shows an example clone genealogy comprises two clone lineages. Clone genealogy can reveal how a cloned code be created, propagated, and vanished during the version updates. In a clone genealogy, each clone group pair with mapping relationship is connected by one high-level clone group evolution pattern or more.

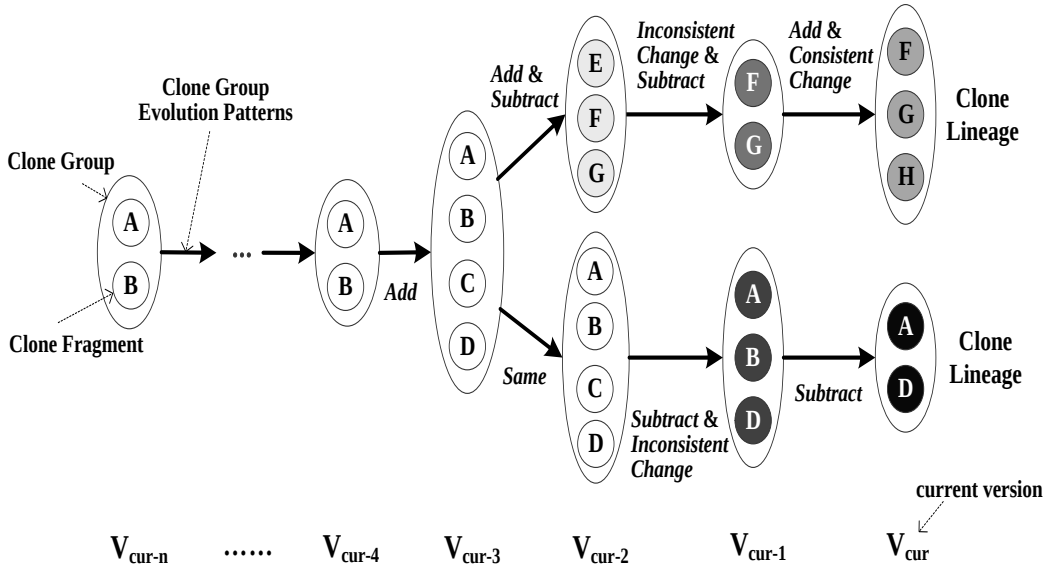

Figure 8. An Example Clone Genealogy

4. Experimental Setup

This section gives a brief overview of the systems we have studied, and the parameters we used for the experiment.

4.1. Subject Systems

Since our previous clone detection study [20] was conducted on C systems, we decided to focus on subject systems written in C. We selected 9 open source C systems which have different size and belong to different domains, such as editor, server, e-mail client and database system. An important basis for selecting these systems is that all of them have multiple stable releases continuously updated, as shown in Table 2. The average sizes of these systems range from approximately 11K to 1003K lines of code (LOC).

Table 2. Subject Systems

Subject System	LOC (min-max, avg)	Duration	No. of Releases	Domain
Bluefish	26027-71232, 51589	2001-01 to 2014-05	18	HTML editor
Claws Mail	158350-344173, 220235	2005-03 to 2014-10	64	Email client
Dovecot	112392-332087, 214683	2008-06 to 2014-05	6	Email server
Emacs	318258-353694, 341986	2005-02 to 2013-03	11	Text editor
FFmpeg	319002-739058, 543560	2013-09 to 2014-07	14	Multimedia framework
Lighttpd	51691-54520, 53378	2007-04 to 2014-03	21	Web server
Memcached	10213-12292, 11133	2011-08 to 2014-10	9	Distributed memory object caching system
PostgreSQL	961348-1058597, 1003498	2011-12 to 2014-03	21	Database system
Wget	17928-88969, 44560	1998-09 to 2014-10	25	File download tool

4.2. Parameters

We set the minimum token length to 30 for FCD, because very short code fragments have not enough practical significance. And the same value was also used in other studies previously [7, 12].

Our experiment regards the latest updated release as current version, tracing going back to find the origin code clones. The accuracy of clone mapping was of great concern. We determined the threshold *t* of *TokenSimilarity* function through a series of contrast experiments, and manually inspect the validity of mapping results, as shown in Figure 9, to clarify there exist little error in the false negative value according to missing check. Finally we set value of *t* to be 0.85 because we want to map clone group pairs as much as possible in premise of low false positive.

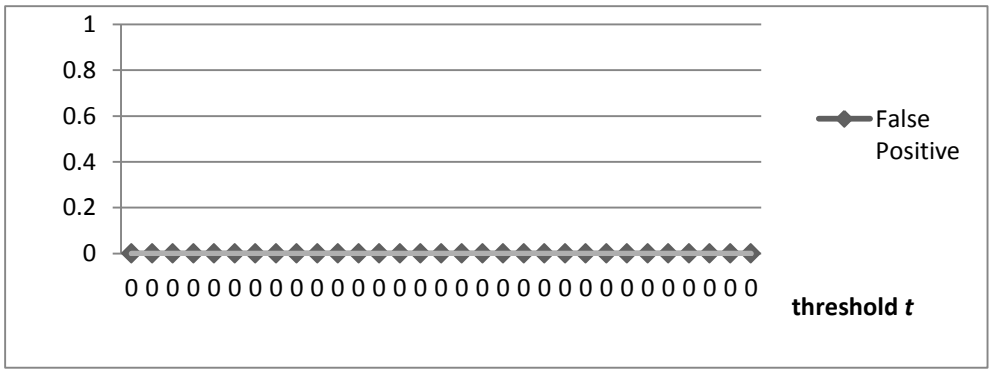

Figure 9. Validity of Clone Mapping with Different Threshold *t*

5. Study Results

In this section we provide the results of our study. At first we analyzed our clone genealogies extraction results and clone group evolution patterns. And then discuss the alive clone lineages in current version.

5.1. Function Clone Genealogies

Since our study was focus on type-1 and type-2 function clones, Table 3 shows the general information of function clone genealogies we extracted using cGen. Configuration of our experimental environment as follow: i3 CPU 2.2GHz, 2GB of memory, operating system is Ubuntu 14.04, runtime of cGen needs to finished all extraction work as listed in Table 3, it mainly depends on size of subject systems and the number of releases used for studied. Empirical study shows that cGen can efficiently extract function clone genealogies from multiple versions of a system. To clarify, we counted the number of genealogies excluding those only compose of one single clone group. For example, a clone group is newly added in one version and soon be removed in the next version. We take no account of this kind of clones because it cannot provide us evolution clue.

Table 3. Function Clone Genealogies Extracted from cGen

Subject System	Total No. of Functions	No. of Genealogies		Size of Genealogies		Runtime (s)
		Total	age>0.5α (percentage)	Total	Avg.	
Bluefish	23562	70	9 (12.86%)	1115	15.93	0.64
Claws Mail	300423	436	175 (40.14%)	51332	117.73	263.64
Dovecot	30179	225	71 (31.56%)	2998	13.32	5.01
Emacs	40597	213	117 (54.93%)	3931	18.46	8.61
FFmpeg	143577	432	227 (52.55%)	13263	30.70	132.05
Lighttpd	21291	75	68 (90.67%)	5004	66.72	5.96
Memcached	3169	7	5 (71.43%)	108	15.43	0.04
PostgreSQL	33364	113	95 (84.07%)	7106	62.88	10.94
Wget	13492	23	8 (34.78%)	643	27.96	0.29

In this study, each clone fragment is exactly a function, and on this foundation, we use the number of functions to measure size of clone genealogies. So in the fourth item of Table 3, size of genealogies means the number of functions that belong to clone genealogies. One clone genealogies may comprise multiple clone lineages, among which clone groups may overlapped in terms of the same origin. We only count one time for overlapping part. Statistic results of long lived function clone genealogies also list in Table 3, here we regard a genealogy as long lived genealogy if it undergo more than half of the number of versions used for studies. The number of releases may vary from system to system, so we set a factor α represent the number of releases for corresponding system. That is, we regard genealogies with age more than 0.5α as long lived genealogies. We found that in most of subjects the percentage of long lived genealogies is more than 30%. For *Lighttpd* and *PostgreSQL*, the proportions of long lived genealogies are even more than 80%. The only exception is *Bluefish*, in which nearly 13% of all genealogies are found undergo more than half releases. One possible reason for this relatively low number of long lived genealogies compared to other may be one of the selected version, *Bluefish* 2.2.0 is a new major release and the start for the 2.2 series. Under the hood *Bluefish* 2.2.0 has a massive number of changes, which lead to evolution clue interrupt.

5.2. Clone Group Evolution Patterns

For each subject system, we statistic the percentage of each clone group evolution pattern occur during the software evolution, as shown in Table 4. Results show that the ratio is not exactly the same for different subject system. It suggests that the scale or domain of system, or programming habits of developers may have some effect on clone evolution. But there is not much difference in general. Overall, however, most of clone did not change during the version upgrade, and both consistent change and inconsistent change are less frequent for all subject systems. As for those changed clones, consistent

change has a higher percentage than inconsistent change. This explains that most clones can be maintained well over the course of evolution. Furthermore, understanding the evolution patterns of clones can help reduce its negative impact on maintenance. For example, pay close attention to clones occurred inconsistent change pattern may detect the potential bugs in system [21, 22].

Table 4. Percentage of each Clone Group Evolution Pattern

Subject System	Same	Add	Subtract	Consistent Change	Inconsistent Change
Bluefish	84.36%	5.90%	5.38%	4.36%	0.00%
Claws Mail	95.34%	2.00%	1.82%	0.76%	0.07%
Dovecot	38.67%	29.51%	26.92%	3.79%	1.11%
Emacs	79.32%	8.20%	9.65%	2.03%	0.80%
FFmpeg	72.57%	12.61%	12.00%	2.42%	0.39%
Lighttpd	96.93%	1.19%	1.33%	0.42%	0.14%
Memcached	94.74%	0.00%	0.00%	5.26%	0.00%
PostgreSQL	80.25%	9.83%	9.79%	0.14%	0.00%
Wget	85.94%	5.73%	5.21%	3.12%	0.00%

5.3. Alive Clone Lineages in Current Version

In practice, people may focus more on code clones in current version. For this purpose, we statistics the number of clone groups (CG) and alive clone lineages (ACL) specific to current version, as shown in Table 5. In addition, we computed age of each clone lineages remain in current version, that is, the number of releases a clone group go through from the initial version to current version.

We found that, most of clone groups in the current version are evolved from previous version. Among the subject systems, more than half of them did not have any new clone group in current version, whereas other systems have a certain number of new created clone groups. This value may response maintenance activity to some extent. If a new release version is mostly a minor bug fix release, it rarely changed clones. However, if it has a massive number of changes, for example, a new major release published or some new features added, it would have an impact on clones in the system, and bring instability to clone evolution.

Table 5. Alive Clone Lineages in Current Version

Subject System	No. of CG		No. of ACL	Age (min -max, avg.)	Avg. age/No. of releases
	Total	New			
Bluefish	48	0	48	2-18, 7.52	41.78%
Claws Mail	459	0	459	4-64, 44.39	69.36%
Dovecot	238	50	188	2-6, 4.28	71.33%
Emacs	60	30	30	3-11, 7.53	68.45%
FFmpeg	396	36	360	2-14, 10.15	72.50%
Lighttpd	74	0	74	2-21, 20.38	97.05%

Memcached	6	0	6	4-10, 8.67	96.30%
PostgreSQL	106	0	106	2-21, 19,18	91.33%
Wget	16	0	16	3-21, 11.19	44.76%
Total	1403	116	1287	**Average**	72.54%

Moreover, the average age of clone lineages in current version is relatively older, exceeds 70% of the total number of selected versions. There exist many clone groups fairly stable, keep alive from the starting version to current version. The proportion of *Bluefish* and *Wget* are relatively low compared to others. We analyzed the reasons and found that maybe long interval between two adjacent versions was one of the main causes of short life. For example, the update interval between the 10th release *Bluefish-1.0.7* and the 11st release *Bluefish-2.0.0* is more than three years, and the update interval between the 12nd release *Wget-1.10.2* and the 13rd release *Wget-1.11* is exceeding two years. This can lead to the clone mapping relationship broken during the update process. Thereby, cause the clone lineages relatively short in current version. Furthermore, different version update branches usually evolve independently, that may also cause the clone evolution clue broken if we choose the version inappropriate. We performed the experiment for *Bluefish* again with different versions, choose *Bluefish-2.2.0* as the starting version, and extracted with eight versions, get the average age of clone lineages in current version to be 6.27, account for 78.38% of the total number of studied versions. The experimental results indicate that choose of versions has important effects on results.

6. Conclusion

In this paper, we implement a function clone genealogies extractor cGen, it traces clone across multiple versions based on its token representation and location attributes. We extend previous studies about clone evolution patterns, present low-level patterns defined for clone fragment, which allow take steps to deriving clone group evolution patterns. Using cGen we performed experiments on 9 C open source system, take all versions of a system in a chronological order as input, analyzed function clone genealogies and clone evolution patterns from the perspective of software evolution. We found that in most of the subjects the percentage of long lived function clone genealogies was more than 30%, and most of clones did change during the evolution and lived a long life, whereas there exist some unstable clones may do harm, so need further study. Specifically, we statistics alive clone lineages in current version, and found that long upgrade interval is almost one of the primary causes of short life of clones. It is necessary to investigate the appropriate selection method for studied versions.

Furthermore, the experimental results can provide useful information about the evolution clue of clones across multiple versions of a system, which contributed to clone management. For example, those stable evolved clones can be considered to be reused, whereas frequently changed clones may cause high maintenance effort or associate risk for introduce defects, need to be refactoring.

Acknowledgements

This work was supported by the National Natural Science Foundation of China under Grant No.61363017 and No.61462071, the Natural Science Foundation of Inner Mongolia under Grant No.2014MS0613, and the Scientific Studies of Higher Education Institution of Inner Mongolia Autonomous Region of China under Grant No.NJZY14039.

References

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research", Technical Report, Queen's university at Kingston, vol. 541, **(2007)**.
[2] J. Krinke, "Identifying similar code with program dependence graphs", Proceedings of the 8th Working Conference on Reverse Engineering, IEEE, **(2001)**, pp. 301-309.
[3] N. Göde and M. Rausch, "Clone Evolution Revisited", Softwaretechnik-Trends, vol. 2, no. 30, **(2010)**.
[4] H. Mei, Q. X. Wang, L. Zhang and J. Wang, "Software analysis: a road map", Chinese Journal of Computer, vol. 32, no. 9, **(2009)**, pp. 1697-1710.
[5] N. Göde and R. Koschke, "Frequency and risks of changes to clones", Proceedings of the 33rd International Conference on Software Engineering, ACM, **(2011)**, pp. 311-320.
[6] J. R. Pate, R. Tairas and N. A. Kraft, "Clone evolution: a systematic review", Journal of Software: Evolution and Process, vol. 25, no. 3, **(2013)**, pp. 261-283.
[7] M. Kim, V. Sazawal, D. Notkin and G. C. Murphy, "An empirical study of code clone genealogies", ACM SIGSOFT Software Engineering Notes. ACM, vol. 30, no. 5, **(2005)**, pp. 187-196.
[8] M. F. Zibran and C. K. Roy, "The road to software clone management: A survey", Tech. Report 2012-03, Department of Computer Science, University of Saskatchewan, Canada, **(2012)**.
[9] N. Göde, "Clone Evolution", Logos Verlag Berlin GmbH, Berlin, **(2011)**.
[10] C. K. Roy, M. F. Zibran and R. Koschke, "The vision of software clone management", Proceedings of the IEEE CSMR-18/WCRE-21 Software Evolution Week, Antwerp, Belgium, **(2014)**.
[11] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: a multi-linguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, vol. 28, no. 7, **(2002)**, pp. 654-670.
[12] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy and K. A. Schneider, "Evaluating code clone genealogies at release level: An empirical study", Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, **(2010)**, pp. 87-96.
[13] Q. Q. Shi, F. J. Meng, L. P. Zhang and D. S. Liu, "Survey of research on code clone technique", Application Research of Computers, vol. 30, no. 6, **(2013)**, pp. 1617-1623.
[14] E. Duala-Ekoko and M. P. Robillard, "Clone region descriptors: Representing and tracking duplication in source code", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 20, no. 1: 3, **(2010)**.
[15] N. Göde and R. Koschke, "Studying clone evolution using incremental clone detection", Journal of Software: Evolution and Process, vol. 25, no. 2, **(2013)**, pp. 165-192.
[16] T. Bakota, "Tracking the evolution of code clones", SOFSEM 2011: Theory and Practice of Computer Science, Springer Berlin Heidelberg, **(2011)**, pp. 86-98.
[17] R. K. Saha, C. K. Roy and K. A. Schneider, "gCad: A near-miss clone genealogy extractor to support clone evolution analysis", Proceedings of the 29th International Conference on Software Maintenance (ICSM), IEEE, **(2013)**, pp. 488-491.
[18] M. Ci, "Research on clone genealogy extracting method based on CRD clone group mapping", Harbin Institute of Technology, **(2013)**.
[19] J. R. Cordy, "The TXL source transformation language", Science of Computer Programming, vol. 61, no. 3, **(2006)**, pp. 190-210.
[20] Q. Q. Shi, L. P. Zhang, L. L. Yin and D. S. Liu, "Based on suffix array detect code clone", Computer Engineering, vol. 39, no. 9, **(2013)**, pp. 123-127.
[21] T. Bakota, R. Ferenc and T. Gyimothy, "Clone smells in software evolution", Proceedings of the 23rd International Conference on Software Maintenance (ICSM), IEEE, **(2007)**, pp. 24-33.
[22] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou and A. E. Hassan, "An empirical study on inconsistent changes to code clones at release level", Proceedings of the 16th Working Conference on Reverse Engineering, IEEE, **(2009)**, pp. 85-94.

Authors

Tu Ying, she was born in 1992, and she is a graduate student at Inner Mongolia Normal University. Her research interests include code clone analysis and clone evolution.

Zhang Li-ping, she was born in 1974. She received her M.S. degree from Inner Mongolia University in 2005. She is an associate professor at Inner Mongolia Normal University. Her research interests include software engineering and software analysis.

Wang Chun-hui, she was born in 1979. She receives her M.S. degree from Inner Mongolia University in 2008. She is a lecturer at Inner Mongolia Normal University. Her research interests include plagiarism detection and code clone analysis.

Liu Dong-sheng, he was born in 1956. He received his B.S. degree from Nanjing University of Science and Technology in 1982. He is a professor at Inner Mongolia Normal University. His research interests include software engineering, software analysis, computers in education, etc.