Evaluating Code Clone Genealogies at Release Level: An Empirical Study

Ripon K. Saha, Muhammad Asaduzzaman, Minhaz F. Zibran, Chanchal K. Roy, and Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Saskatoon, SK, Canada S7N 5C9
{ripon.saha, md.asad, minhaz.zibran, chanchal.roy, kevin.schneider}@usask.ca

Abstract

Code clone genealogies show how clone groups evolve with the evolution of the associated software system, and thus could provide important insights on the maintenance implications of clones. In this paper, we provide an in-depth empirical study for evaluating clone genealogies in evolving open source systems at the release level. We develop a clone genealogy extractor, examine 17 open source C, Java, C++ and C# systems of diverse varieties and study different dimensions of how clone groups evolve with the evolution of the software systems. Our study shows that majority of the clone groups of the clone genealogies either propagate without any syntactic changes or change consistently in the subsequent releases, and that many of the genealogies remain alive during the evolution. These findings seem to be consistent with the findings of a previous study that clones may not be as detrimental in software maintenance as believed to be (at least by many of us), and that instead of aggressively refactoring clones, we should possibly focus on tracking and managing clones during the evolution of software systems.

1. Introduction

Programmers often copy code fragments and then paste them with or without modifications during software development. Such duplicated code fragments are known as software clones or code clones. Previous studies have shown that systems contain duplicate code in amounts ranging from 5-15% of the code-base [23] to as high as 50% [22]. Despite their usefulness [12, 15], the presence of identical or near identical code fragments may add to the difficulties of software maintenance. For example, if a bug is detected in a code fragment, all the fragments similar to it should be investigated to check for the same bug and when enhancing or adapting a piece of code, duplicated fragments can multiply the work to be done [19]. Code clones are also considered as one of the bad smells of a software system [3, 10]. Consequently, identification and management of software clones has now become an essential part of software maintenance. However, due to the intense use of template-based programming [12], a certain amount of clones are likely acceptable.

Previous studies were highly influenced by the idea that clones are harmful and can be removed through refactoring [15]. This notion has been challenged by the work of Kim et al. [15]. They provided a clone genealogy model and analyzed the clone genealogies of two open source software systems. While a clone group consists of a set of code fragments in a particular version of a software that are clones to each other, a genealogy of a clone group describes how the code fragments of that clone group propagate during the evolution of the subject system. Each clone genealogy consists of a set of clone lineages that originate from the same clone group (source). A clone lineage is a directed acyclic graph that describes the evolution history of a clone group from the beginning to the final release of the software system. The empirical study described by Kim et al. on code clone genealogy reveals that clones are not always harmful. Programmers intentionally practice code cloning to achieve certain benefits [12, 13]. During the development of a software system, many clones are short lived. Refactoring them aggressively can overburden the developers. Their study also shows that many long-lived consistently changing clones are not locally refactorable. Such clones cannot be removed from the system through refactoring [15].

We are motivated by the work of Kim et al. [15]. They were the first to analyze clone genealogies. However, they only analyzed two small Java systems. They also speculated that the selected systems might not have captured the characteristics of larger systems and thus, further empirical evaluations need to be carried out for larger systems of different languages. After Kim et al. several other researchers also investigated the maintenance implications of clones. Kapser and Godfrey [12] conducted several studies in the area and showed that clones might not always be harmful and even could be useful in a number of ways. Krinke [16, 17] studied change types and the stability of code clones based on the changes between the revisions of several open source systems. Although he analyzed several systems written in C, C++ and Java, he did not focus on evaluating clone genealogies. Bettenburg et al. [5] analyzed inconsistent changes of code clones to determine their contribution to software defects. They also noted the importance of a release level empirical study compared to that at the revision level. However, to the best of our knowledge, no further extensive empirical evaluations have been carried out to examine the code clone genealogies with different languages or variable program sizes.

In this paper, we followed the footsteps of Kim et al. [15] by conducting an in-depth empirical study on the evaluation of clone genealogies in 17 open source systems covering four popular programming languages, C, Java, C++ and C#. However, unlike Kim et al. [15], we did not work at the revision level; rather, we analyzed the evolution of clones at the release level since they are less affected by short term experimentations of the developers in the software development process [5]. The systems are selected from different areas and have rich development histories. In particular, we focus on the following two research questions:

(1) How do the clone genealogies look like in open source software written in different languages and of different sizes with variable release histories?

(2) Do clone genealogies at the release level share any common quantitative characteristics, and do any particular type of genealogies exhibit higher longevity than the others?

With an extensive study of 17 open source systems written in four different languages, we have reached the following conclusions:

(1) Most of the clone groups are propagated through subsequent releases either without any changes or with changes only in identifier renaming. Many of them reach to the final releases of the subject systems and contribute to the number of alive genealogies. We have found that, on average about 67% of the genealogies among all systems do not have any addition or deletion of lines or any syntactic changes. Moreover, an average of roughly 69% of these syntactically similar genealogies reach to the final releases.

(2) We have observed that from about 11% to 38% of the genealogies are changed consistently over the entire course of the evolution.

(3) Among the dead genealogies, many of them are removed within a few releases.

(4) Clone evolution is not highly affected by development languages or project sizes.

The rest of the paper is organized as follows. Section 2 outlines the study approach. In Section 3, we describe the experimental setup and then present the results of the case study in Section 4. Section 5 describes the threats to the validity of our study and in Section 6 we discuss some other studies related to ours. Finally, Section 7 concludes the paper with our future plans.

2. Study Approach

Our primary objective is to study how code clones evolve over different releases during system evolution in terms of the clone genealogy. In addition to this, we also want to investigate whether the findings by Kim et al. [14, 15] based on two small Java systems also hold for other systems of diverse varieties, varying system sizes and systems written in different programming languages. Our objective is not to validate the findings of Kim et al. by replicating the same experiment with exactly the same settings, rather we wanted to examine how code clones evolve in software systems of varying sizes written in different programming languages using their clone genealogy model. Thus, we develop a clone genealogy extractor similar to theirs except that the location overlapping function is replaced by a snippet matching algorithm. Kim et al. developed a *diff* based location tracker that maps the line numbers of a snippet to its old line numbers in the previous release. They also discussed that the location overlapping function did not work well when lines are modified or reordered in a file because *diff* cannot capture such changes. The purpose of the location overlapping function was to find out the exact mapping of a clone group from the previous release to the next. To fulfill the same objective we have developed a location independent approach, snippet matching function that maps a clone group from the previous release to its next based on identifier matching. The following paragraph discusses how our modified Clone Genealogy Extractor (CGE) works.

2.1. Clone Genealogy Extractor

Our clone genealogy extractor automatically extracts clone genealogies across the releases of a program. The steps are summarized as follows: (1) first, we collect multiple releases of a program and then sort them in chronological order; (2) second, we run CCFinderX on all these releases with a batch processor; (3) third, we collect the clone group information on each release produced by CCFinderX; and (4) finally, the output and the intermediate files generated by CCFinderX are then used as input for the CGE.

In order to map clone groups of successive releases, the CGE uses both *TextSimilarity* and *SnippetMatching* functions as described below. The CGE maps clone groups based on the highest text similarity and snippet matching scores. If the highest text similarity score is different from the highest snippet matching score, the heuristic selects both of them in order to avoid ambiguity. The following subsections describe the *TextSimilarity* and *SnippetMatching* techniques.

2.2. Text Similarity

The text similarity between two code snippets C_1 and C_2 is determined by calculating the common tokens sequence with respect to their token sizes. By considering tokens generated by CCFinderX, we count the textual matches across releases. Equation (1) below describes the *TextSimilarity* function. Here $|C_1|$ and $|C_2|$ are the token sizes of code snippets of C_1 and C_2 respectively. $|C_1 \cap C_2|$ is the size of common ordered tokens between C_1 and C_2, calculated using the longest common subsequence (LCS) algorithm. In order to have consistency with Kim et al., we used a text similarity heuristic of 0.3. With this similarity threshold, the length and size of the genealogies are neither overestimated nor underestimated [15].

$$TextSimilarity(C_1, C_2) = \frac{2|C_1 \cap C_2|}{|C_1| + |C_2|} \qquad (1)$$

2.3. Snippet Matching

By applying the text similarity heuristic, we can eliminate many uninteresting mappings that are not syntactically similar. However, the text similarity score itself is not always enough to get better result. In snippet matching, on the other hand, we match the snippets based on the similarity of identifiers. The text similarity function produces a higher value than the given threshold for all of the mappings that are syntactically similar. However, in such cases, it is highly probable that they have different identifier names. The snippet matching algorithm is applied on all the mappings produced by the text similarity function above. The algorithm takes two code fragments and produces a value between 0 and 1 to reveal how much these snippets are identical by their identifier names. We first extract the identifiers from each of the snippets and then apply LCS algorithm on them to find the matching score as follows:

$$SnippetMatching(S_i, S_j) = \left\{ \frac{LCS(IS_i, IS_j)}{len(IS_i)} + \frac{LCS(IS_i, IS_j)}{len(IS_j)} \right\} / 2 \qquad (2)$$

where, IS_i = {set of identifiers of snippet, S_i}, IS_j = {set of identifiers of snippet, S_j}, and $LCS(IS_i, IS_j)$ = {Longest common subsequence for the identifiers of the snippets S_i and S_j}.

It is possible that some of the identifiers might be common between two code snippets of two different clone groups of two successive releases, but it is unlikely that they maintain the same sequence and produce a higher similarity value. Again, it is possible that some identifiers might be renamed in the next release. In such cases, the same snippets in two releases might produce very low snippet matching similarity value. To overcome such situations, we calculate the snippet matching values for all possible pairs between two clone groups of two successive releases and take the one with the maximum similarity value. There is a threat to this approach in the cases where all the identifier names of all snippets in the same clone group are changed/renamed in the next release. However, in our experience, such a situation is very unlikely to occur.

Fig. 1 represents a clone genealogy that consists of three clone lineages marked with different line styles. All the three lineages evolve from the same clone group that consists of three code snippets (A, B, C) and is called the source of the lineages. Each clone lineage describes how a sink node evolves from the source node. For example, the sink of one of the clone lineages that consists of two code snippets (E, G) evolves from the source node with addition and inconsistent changes, subtraction and inconsistent change, addition and consistent changes evolutions patterns through the release history. Thus, a clone genealogy captures the evolution of a clone group through the release history, and all the lineages that belong to a clone genealogy originated from that clone group.

For each system, we have collected the total number of genealogies including the number of alive and dead genealogies. By *alive genealogies* we mean the genealogies of which at least one lineage reaches to the final release. On the other hand, if none of the lineages of a genealogy reaches to the final release, we call that genealogy as a *dead genealogy*. We then study what proportion of the genealogies are changed consistently and what proportion of them remain syntactically the same.

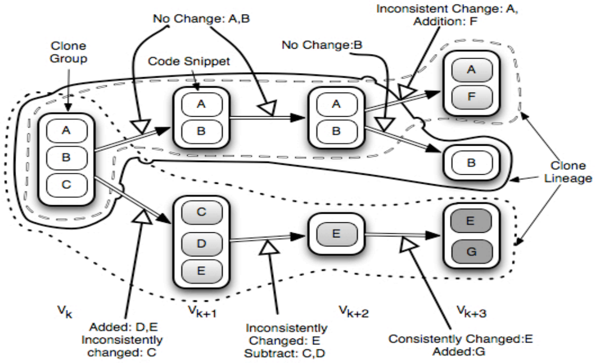

No Change: A,B
Inconsistent Change: A, Addition: F
Clone Group
Code Snippet
No Change:B
A
F
A
B
A
B
A
B
C
B
Clone Lineage
C
D
E
E
E
G
V_k
Added: D,E Inconsistently changed: C
V_k+1
Inconsistently Changed: E Subtract: C,D
V_k+2
V_k+3
Consistently Changed:E Added:G

Figure 1. Clone genealogy

3. Experimental Setup

In this section we provide a brief overview of the systems we have studied, and the clone detection tool we used for the experiment.

3.1. Subject Systems

We studied 17 open source software systems [6, 26] covering four different programming languages, C, C++, Java and C# as shown in Table 1. The sizes of these systems range from approximately 9K to 204K source lines of code (SLOC), excluding comments and blank lines. The systems are selected from different domains such as text editor, email client, graphics library, test framework and so on.

3.2. Clone Detection

We used the AIST CCFinderX [7] to detect code clones in each release. CCFinderX is a major revision of CCFinder [11]. CCFinderX is instructed to detect clones with TKS (minimum number of distinct types of tokens) set to 12 (default setting). In order to detect clones of large enough for practical significance, we set the minimum token length to 30. The same value for the minimum token length was also used in other research projects in the past [15].

Table 1. Subject systems

Lang	Subject System	SLOC	Duration	No. of Releases
Java	JUnit	2,179-8,785	2003-05-12 to 2009-12-08	20
	CAROL	2,812-11,694	2002-11-12 to 2005-04-13	10
	dnsjava	11,025-23,334	2001-03-29 to 2009-11-21	22
	JabRef	11,352-74,104	2003-11-30 to 2010-04-14	33
	iText	51,860-82,164	2002-03-07 to 2008-01-25	49
C++	KeePass	14,789-43,644	2003-11-17 to 2006-10-14	35
	Notepad++	26,937-81,980	2003-11-25 to 2007-02-04	30
	7-Zip	71,638-100,823	2003-12-11 to 2009-02-03	45
	eMule	6,803-203,780	2002-07-07 to 2010-04-07	73
C	Wget	14,209-40,021	1998-09-23 to 2009-09-22	17
	Conky	7,029-42,060	2005-07-20 to 2010-03-30	70
	ZABBIX	12,468-70,890	2004-03-23 to 2010-01-27	28
	Claws Mail	126,247-203,783	2005-03-19 to 2010-01-31	47
C#	NAnt	686-52,533	2001-07-19 to 2007-12-08	22
	iTextSharp	33,545-163,890	2003-02-04 to 2007-03-08	26
	Process Hacker	10,349-123,878	2008-10-17 to 2010-01-23	38
	ZedGraph	2,439-26,433	2004-08-02 to 2008-12-12	28

4. Study Results

This section presents the results of our study. Since the subject of this paper is code clone evolution in terms of the clone genealogy, at first we characterize different types of genealogies and then discuss our findings pertaining to them.

4.1. Clone Genealogies

This subsection characterizes the evolution of clone groups in terms of genealogies. We will focus on four types of genealogies, (1) alive genealogy, (2) dead genealogy, (3) syntactically similar genealogy, and (4) consistently changed genealogy in order to discuss the evolution characteristics. A genealogy is called *alive genealogy* (AG), if it contains at least one clone group up to the final release; otherwise, it is marked as a *dead genealogy* (DG). The term *syntactically similar genealogy* (SSG) refers to those genealogies in which the clone groups are propagated through subsequent releases either without any changes or with changes only in formatting and identifiers (e.g., renaming of identifiers) in their code snippets. No lines are added or deleted in the snippets. However, cloned snippets could be moved from one location to another in the same file of the subsequent releases. *Consistently changed genealogy* (CCG) means genealogies in which all the clone groups have at least one consistently changed pattern of any sort (e.g., addition of a new line to all the snippets of the clone groups). Table 2 presents the total number of genealogies and the proportions of the four types of genealogies mentioned above.

From Table 2 we see that the proportions of alive and dead genealogies are not largely affected by programming languages or program sizes. For Java, C and C++ systems, the values are very close. The proportions of alive genealogies of these systems vary from 69% to 72% whereas C# systems contain almost 76% of alive genealogies, the highest among the four languages. On the other hand, when we examined the subject systems in terms of program size (Table 3) we can see that in general, the average proportions of alive genealogies increased with the increase of program size. It means more genealogies disappeared from the smaller systems compared to that of the larger ones, which suggests that perhaps clones are more manageable in systems with a smaller size

Table 2. Clone genealogies

System	Total # of Gen.	AG (%)	DG (%)	SSG (%)	CCG (%)
JUnit	127	78.74	21.26	81.89	15.75
CAROL	141	44.68	55.32	56.73	38.30
dnsjava	417	82.97	17.03	85.37	12.23
JabRef	1132	73.41	26.59	66.25	26.06
iText	1568	68.75	31.25	74.62	20.22
Avg. of Java Systems		71.43	28.57	72.67	21.77
KeePass	790	70.76	29.24	73.54	20.63
Notepad++	977	81.99	18.01	73.69	19.86
7-Zip	1427	65.38	34.62	64.62	24.46
eMule	3547	66.08	33.92	59.57	29.86
Avg. of C++ Systems		68.79	31.21	64.32	26.18
Wget	206	57.77	42.23	60.68	28.64
Conky	1328	53.69	46.31	82.45	11.97
ZABBIX	1026	65.79	34.21	49.31	28.65
Claws Mail	2363	85.57	14.43	63.26	27.08
Avg. of C Systems		71.68	28.32	65.43	23.40
NAnt	625	76.96	23.04	55.20	34.08
iTextSharp	2666	77.16	22.84	86.38	10.43
Process Hacker	950	71.79	28.21	73.26	20.32
ZedGraph	374	76.74	23.26	62.83	24.87
Avg. of C# Systems		76.00	24.00	77.55	16.84
Avg. of all Systems		70.33	29.67	66.56	24.28

Table 3. Distribution of genealogies by program size

Program Size	AG (%)	DG(%)	SSG(%)	CCG (%)
<50K	64.65	33.35	76.15	17.71
50K-100K	71.04	28.96	65.32	24.60
>100K	74.58	25.42	69.36	22.78

compared to a larger one. Thus, a clone tracking and maintenance tool might be more effective for larger systems. In the following subsections we will have a closer look at the four types of genealogies.

4.2. Consistently Changed Genealogies

From Table 2 we see that the number of consistently changed genealogy varies from 10.43% to 38.30% for the subject systems. The average number of consistently changed genealogies varies in terms of program size (17.71% to 24.6%) or implementation language (16.84% to 26.18%). As we see the variations are not too drastic and do not reveal any systematic change pattern. However, from our study we see that the number of consistently changed genealogies is not very high (on average 24.28%).

Among the subject systems, *CAROL* and *dnsjava* were analyzed by Kim et al. [15]. Even though they studied at the revision level and we studied at the release level, we observed a similar proportion of consistently changed genealogies in *CAROL*. However, there is a bit difference in the number of genealogies detected. They found 122 genealogies from which 13 were eliminated due to template based programming, whereas, we found 141 genealogies. It should be noted that we did not consider template based programming because we believe that such clones are nevertheless clones. Moreover, we have considered release level candidates and applied a combination of snippet matching and text similarity algorithms (discussed earlier). For *dnsjava,* on the other hand, we experienced a significant difference from them. Possible reasons could be that Kim et al. [15] considered revisions until November 2004 whereas we studied releases until November 2009, and some major changes took place in the code-base of *dnsjava* in May 2005. This might have caused many new clones, and most of the new clone groups were propagated to the final release contributing to the higher proportion of alive genealogies.

4.3. Alive Genealogies

In this study, we have found that a substantial proportion of genealogies of all systems are alive, which is 70.33% of total genealogies on average (Table 2). For example, out of 3547 genealogies in *eMule*, 2344 have at least one clone group in the final release, thus about 66% of total genealogies in *eMule* are counted as alive. For *dnsjava*, *Notepad++*, and *Claws Mail* the proportions of alive genealogies are even more than 80%. The only exception is *CAROL*, in which nearly 45% of all genealogies are found alive. The *CAROL* project is now closed and a lot of refactoring was done in the final release [6], which is probably a reason for this relatively low number of alive genealogies compared to others.

One possible reason behind this large number of alive genealogies is that a significant number of clone groups were created in just a couple of releases prior to the final release, and they are counted as alive since it is unknown when they will be removed in the future releases. Table 4 presents the total number of alive genealogies, genealogies that are created within final five releases and the alive genealogies that survive more than half of the release histories for each system. From the table we can get a fairly complete picture of alive genealogies including their lifetimes.

The numbers vary across subject systems possibly due to variable lengths of release histories we have considered. However, for most systems, recently created alive genealogies are not negligible (on average 23.11% for all subject systems within five releases) and a large proportion of alive genealogies survive for more than half of the release histories (47.57% on average). Many of the recently created alive genealogies might or might not be continued in later releases. However, this dualism indicates the importance of incorporating language specific IDE

Table 4. Alive genealogies

System	AG	AG created within recent five releases	AG that survive more than half of release histories
JUnit	100	31 (31%)	68 (68%)
dnsjava	346	15 (4.34%)	82 (23%)
CAROL	63	47 (74.60%)	17 (26.98%)
JabRef	831	72 (8.66%)	400 (48.13%)
iText	1078	490 (45.45%)	765 (70.96%)
KeePass	559	163 (29.16%)	241(43.11%)
Notepad	801	59 (7.36%)	587 (73.3%)
7-Zip	933	34 (3.64%)	678 (72.66%)
eMule	2344	385 (16.42%)	365 (15.57%)
Wget	119	16 (13.44%)	74 (62.18%)
Conky	713	490 (68.74%)	136(19.07%)
ZABBIX	675	11 (1.62%)	467 (69.18%)
Claws Mail	2022	46 (2.29%)	1335(66.02%)
NAnt	481	275 (57.17%)	62 (12.89%)
iTextSharp	2057	864 (42%)	1094 (53.18%)
Process Hacker	682	236 (34.60%)	158 (33.17%)
ZedGraph	287	23(8.01%)	174 (60.63%)
Avg. of all systems		23.11%	47.57%

based clone evolution tracker that may assist managing clones instead of applying refactoring aggressively immediately when clones are encountered.

4.4. Syntactically Similar Genealogies

We further investigate what proportion of clone genealogies remains syntactically the same throughout the evolution. It is important to study such SSGs because clone groups of these genealogies seem stable during the evolution, and thus one may not need any extra care for them (because where there is probability of change, there is a fear of inconsistent changes). Thus, aggressively refactoring them might not be worthwhile. We have noticed that an enormous proportion of clone genealogies are syntactically similar, and on average 66.56% of all the subject systems (Table 2). The highest proportion of syntactically similar genealogies is found in *iTextSharp*, roughly 86%, whereas the lowest is nearly 50% for *ZABBIX*. If we look at them by language (Table 2) we see that the numbers of such genealogies in C and C++ systems are lower than the systems of the other two languages. About 64.32% and 65.43% of genealogies are syntactically similar for C++ and C systems respectively whereas for the systems of the other two languages the value varies from 72.67% to 77.55%. We also noticed variations in terms of program sizes (Table 3). In particular, systems with sizes ranging from 50K to 100K LOC show fewer syntactically similar genealogies compared to the systems of the other two size ranges.

We further examine whether there are any relationships between these syntactically similar genealogies and alive genealogies. From Table 5, we notice that on average 69.04% of syntactically similar genealogies reached to the final releases of the subject systems. On the other hand, on average about 66.61% of alive genealogies did not change syntactically throughout their entire lifetimes. These indicate that most of the clone groups that do not change syntactically are unlikely to be removed during the evolution of the software systems. SSGs are cost-effective in the sense that they require little or no maintenance effort. Instead of aggressively refactoring them, we may track the evolution of such clones so that we can differentiate them from other types of genealogies, those may require more care. In terms of program size, the proportion of syntactically similar alive genealogies over SSG increases with the increase of program size (Table 6). It means more SSGs were propagated to the final releases in larger systems than those of smaller ones. This implies that possibly for smaller systems developers can handle clones more effectively than that for larger ones. However, no strong change relationship was observed for the proportions of alive SSGs over the total number of alive genealogies.

Table 5. Syntactically similar genealogies

System	Alive SSG	% of alive SSG of total SSG	% of alive SSG of total AG
JUnit	85	81.73	85.00
CAROL	34	42.50	53.97
dnsjava	310	87.08	89.60
JabRef	530	70.67	63.78
iText	811	69.32	75.23
Avg. of Java Systems		71.95	73.20
KeePass	402	69.19	71.91
Notepad++	568	78.89	70.91
7-Zip	586	63.56	62.8
eMule	1262	59.73	53.83
Avg. of C++ Systems		64.99	60.77
Wget	60	48.00	50.42
Conky	590	53.88	82.74
ZABBIX	279	55.13	41.33
Claws Mail	1220	81.60	60.33
Avg. of C Systems		66.72	60.90
NAnt	235	68.12	48.86
iTextSharp	1734	75.29	84.30
Process Hacker	505	72.56	74.04
ZedGraph	175	74.47	60.98
Avg. of C# Systems		74.02	75.53
Avg. of all Systems		69.04	66.61

Table 6. Syntactically similar genealogies by program size

Program Size	% of alive SSG of total SSG	% of alive SSG of total AG
<50K	64.29	75.72
50K-100K	68.18	62.70
>100K	71.45	66.44

4.5. Dead Genealogies and Volatile Clones

We were also interested to see how long dead genealogies survive in the systems in terms of the number of releases. For this purpose, we used the term *k-volatile genealogy*, which refers to a dead genealogy that disappears within *k* versions.

To visualize this scenario, we used the same approach defined by Kim et al. [15] as follows:

Let, *f(k)* denotes the number of genealogies with age *k*, $f_{dead}(k)$ denotes the number of dead genealogies with age *k*, $CDF_{dead}(k)$ denotes the cumulative distribution function of $f_{dead}(k)$ and it is the ratio of *k-volatile* genealogies among all dead genealogies. $R_{volatile}(k)$ denotes the ratio of *k* volatile genealogies among all genealogies in a system.

Fig. 2(a-d) represents $CDF_{dead}(k)$ and $R_{volatile}(k)$ for the largest and smallest subject systems for each of the language categories. Here, the horizontal axes represent the ages of the genealogies in terms of releases and vertical axes represents the values of $CDF_{dead}(k)$ or $R_{volatile}(k)$.

Figs. 2(a) and 2(c) represent the $CDF_{dead}(k)$ and $R_{volatile}(k)$ for the largest systems of each of the language categories respectively. The largest Java system is *iText*. We can see from the graph that for this system, 16% of all dead genealogies (5% of all genealogies) disappeared within six releases. In *Claws Mail* (largest C system), 28% of all dead genealogies (5% of all genealogies) disappeared within five releases, and within 10 releases roughly 50% of all dead genealogies (7% of all genealogies) disappeared. For *eMule* (largest C++ system), 33% of all dead genealogies disappeared within only five releases. For the largest C# system, *iTextSharp* we found that the initial value for $CDF_{dead}(k)$ and thus also $R_{volatile}(k)$ to be smaller compared to the other systems. The possible reason behind this difference is that a higher number of dead genealogies (in total 382) span over 19 releases, which is more than 50% of all dead genealogies.

The same attributes for the smallest systems of each language categories are provided in Figs. 2(b) and 2(d). The smallest Java system in our study is *JUnit*. We found that all the dead genealogies (about 21% of all genealogies) of this system disappeared within six releases from when they were created.

KeePass Password Safe is the smallest C++ system with 43K LOC in its final version. Among the dead genealogies for this system, 12% disappeared within five releases. The smallest C system, *Wget* also shows a similar trend but with a much higher ratio. In this particular scenario, 60% of all dead genealogies (25% of all genealogies) disappeared within only six releases and about 97% of all dead genealogies (40% of all genealogies) disappeared within 10 releases. When we plot the same attribute for *ZedGraph* (smallest C# system), we found that this system maintains a similar trend (12% of all genealogies and approximately 52% of all dead genealogies disappeared within five releases).

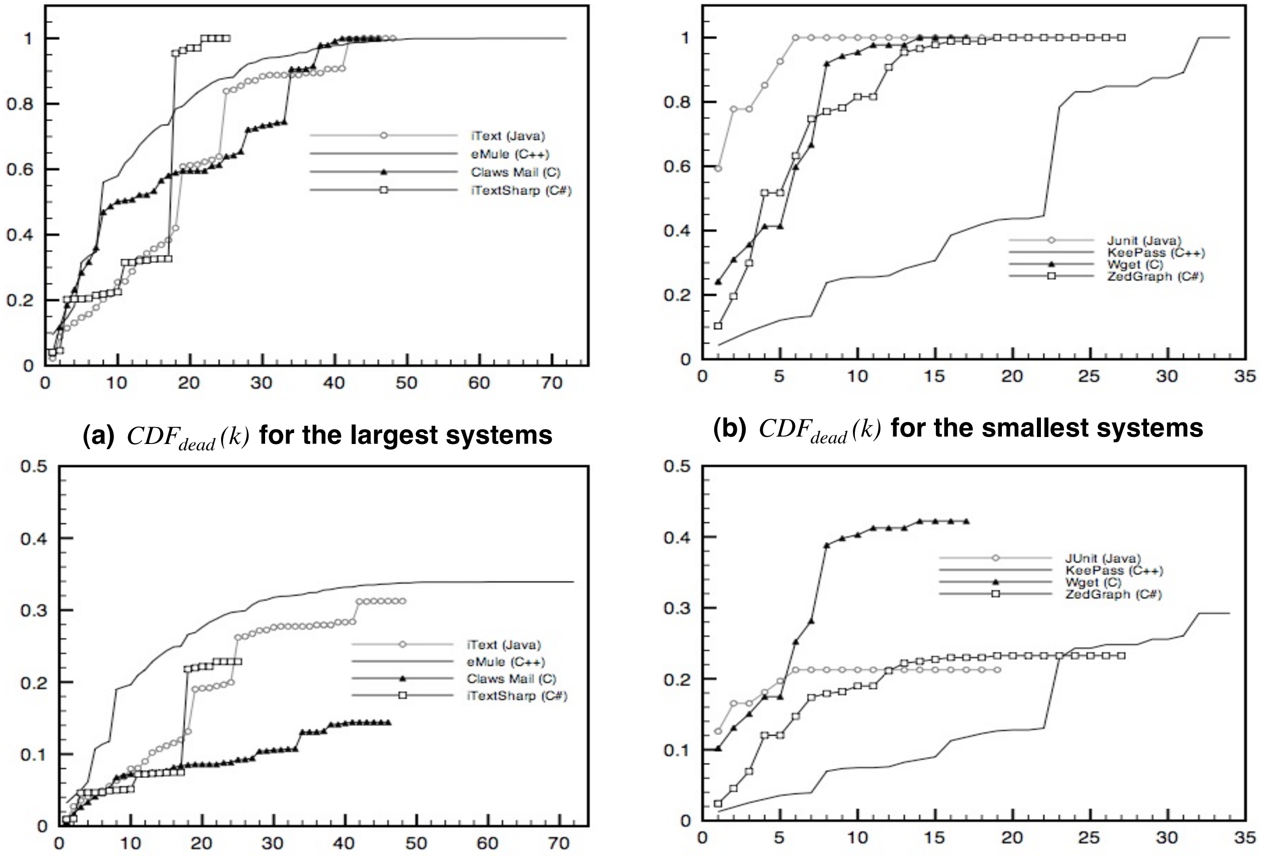
The above data did not reveal any systematic relationship between $CDF_{dead}(k)$ and $R_{volatile}(k)$ for the language categories. However, we have found that even at the release level, the number of volatile clones was not negligible. Moreover, many of them propagate through subsequent releases without any changes. These findings indicate that aggressive refactoring is possibly not a cost-effective solution for such clones and may call for alternative measures such as tracking and managing them in their evolution.

5. Threats to Validity

One of the major threats to this study is that the clone detector we used might have missed certain clones in the systems (false negatives) or detected clones that are not clones in practice (false positives). We used CCFinderX with settings (minimum token length of 30 and minimum token set size of 12) that allow it to detect clones of reasonable size. Although with this setting, some clones might have been missed or some false positive clones might have been considered, we have chosen to use CCFinderX in our study to be consistent with the study of Kim et al. [15] since one of our research objectives was to investigate whether software systems of different languages and of different sizes and varieties show similar trends at the clone genealogy level to that observed by Kim et al. Moreover, CCFinder is recognized as a state of the art clone detector having high recall, although its precision is lower than some other tools [4].

A major part of this study is to map the clone groups from one release of a system to the next for extracting clone genealogies. While we have manually verified all the clone genealogies of some small systems, it was very difficult to manually verify the genealogies for all the systems. In our experience, although we did not find any false positive mappings (at least within our given settings and heuristics) except a few due to CCFinder finding false positive clones, we cannot guarantee that there are no false positive mappings in the results.

Another threat to this study is the limited number of samples. However, to our knowledge this is the first study on the maintenance implications of clones, and in particular on evaluating clone genealogies that considers 17 open source systems of different



(a) $CDF_{dead}(k)$ **for the largest systems**

(b) $CDF_{dead}(k)$ **for the smallest systems**

(c) $R_{volatile}(k)$ **for the largest systems**

(d) $R_{volatile}(k)$ **for the smallest systems**

Figure 2. $CDF_{dead}(k)$ **and** $R_{volatile}(k)$ **for the largest and smallest systems of each language category**

languages of diverse varieties. Since all the systems in our study are open source, one may argue that a similar study on industrial systems may produce different results.

6. Related Work

Studying the evolution of clones is not a new topic and there have been several such studies. While they differ significantly in many aspects, they are also related to this study. Lagüe et al. [18] studied the evolution of clones with six versions of a large telecommunication software system and concluded that although a significant number of clones were removed during the evolution, the overall cloning density increased over time. Antoniol et al. [1] and Li et al. [19] studied the evolution of the Linux kernel and observed that although clone coverage increased early in the development, it stabilized over time. Our study differs from theirs by addressing how code fragments of a clone group change with respect to the other fragments of that group during system evolution.

In recent years, studying the maintenance implications of clones, which is also one of the objectives of our study, has become an active research topic. Kapser and Godfrey [12] conducted large-scale empirical studies and concluded that clones are not necessarily harmful and found several patterns of clones that could be useful in many cases. Juergens et al. [10], on the other hand, argued that unintentionally created inconsistent clones always leads to faults, and concluded that clones could be harmful in software maintenance. While we also studied the maintenance implications of clones, our study significantly differs from theirs in the sense that they did not study the evolution of clones.

Krinke [16] analyzed many revisions of five open source software systems and found that half of the changes to code clone groups are inconsistent and that corrective changes following inconsistent changes are rare. In another study [17], he found that cloned codes are more stable than non-cloned codes and thus require less maintenance effort compared to non-cloned code. Our study differs from his in that we work on releases instead of revisions, and that we particularly focus on evaluating clone genealogies.

Bettenburg et al. [5] studied the inconsistent changes of clones at the release level. They noted that the number of defects through inconsistent changes is possibly substantially lower at the release level than at the revision level. They reported that many clones are created during the software development process due to the experimentation of developers, which the developers can manage well. Thus they worked at the release level instead of the revision level. In order to avoid the affect of such short-term clones, we also choose to work at the release level. However, while they focus on finding the relation of inconsistent changes to software defects for two open source systems, we particularly focus on evaluating clone genealogies using 17 open source systems written in four different languages.

Lozano et al. [20, 21] conducted several studies on the maintenance implications of clones. While they could not find any systematic relationships between cloning and maintenance efforts, they concluded that change efforts might increase for a method when it has clones. Although the underlying clone detection tool is the same as ours, their approach is different from ours in many aspects. In particular, they work on the revision level, whereas we work on the release level and that they focus on the changes at the function level, whereas we focus on the clone level itself. Moreover, they studied only Java systems, which might have also affected the findings.

Göde [9] proposed a computationally efficient approach that models type-1 (identical code fragments except for variations in whitespace and comments) clone evolution based on the source code changes made between consecutive program versions of several open source systems. While he concluded that the ratio of clones decreased in the majority of the systems and cloned fragments survived more than a year on average, no general conclusion on the consistent or inconsistent changes to clone groups was proposed. Our work differs from his in several ways. In particular, he used an incremental clone evolution model and only considered type-1 clones whereas we considered both type-1 and type-2 (where syntactically similar fragments are also considered clones) clones, and that he worked at the revision level whereas we worked at the release level. Bakota et al. [3] proposed a machine learning approach for detecting inconsistent clone evolution situations and found different 'bad smells' using twelve versions of Mozilla Firefox. However, they studied the evolution patterns of cloned fragments whereas we studied clone groups, and they worked at the revision level (and only 12 monthly revisions of Mozilla Firefox) whereas we studied release versions of many systems written in different languages.

Thummalapenta et al. [27] performed an empirical evaluation on four open source C and Java systems for investigating to what extent clones are consistently propagated or independently evolved. While they focused on identifying evolution of cloned codes over time and relating the evolution pattern with other parameters (clone granularity, clone radius and cloned code fault-proneness), we focus on evaluating clone genealogies with 17 open source software systems covering four popular programming languages.

The most closely related work to ours is the study of Kim et al. [15], which is also one of the motivations of our study. However, they studied only two small Java systems and at the revision level. On the other hand, we studied at the release level and with 17 diverse varieties of open source systems written in four different programming languages. Furthermore, instead of location mapping, we have used snippet matching together with text similarity for mapping the clone groups from one version to the next. This allows us to map clone groups even when lines are modified or reordered in the next version. Aversano et al. [2] extended the clone evolution model of Kim et al. [15] by grouping inconsistent changes to independent and late evolution classes. Again, they studied only two open source Java systems namely *ArgoUML* and *dnsjava* and reported contradictory findings for the consistently changed clone groups.

7. Conclusion

In this paper, we have presented an empirical study for evaluating code clone genealogies using 17 diverse categories of open source software systems written in four different programming languages. We have set up our experiment based on the genealogy model of Kim et al. [15] and extended their empirical study in different dimensions. While Kim et al. concentrated on the consistently changed genealogies, and the nature of volatile clones by analyzing two small Java systems, we attempted to draw a more detailed picture of clone genealogies by analyzing a larger number of systems, and systems written in different development languages, systems of varying size, and systems with varying development histories. Kim et al. found that (at the revision level) from 36% to 38% of genealogies were changed consistently, whereas we have found that (at the release level) from 11% to 38% of genealogies were changed consistently, which does not seem contradictory. Again, they reported that volatile clones were disappearing within a short time from the systems and noted that from 48% to 72% of volatile clones were disappearing within eight check-ins. We also found that even at the release level many volatile clones disappear within a few releases. In addition, our study reveals some other interesting characteristics of code clone genealogies. We have found that for all subject systems, many genealogies are alive and long-lived, which implies that more clone groups are created than those that are removed. In most of the genealogies for the subject systems, clone groups are propagated through releases either without any change or with changes just in identifier renaming. Hence, it is possible that these types of genealogies do not need any extra care during software maintenance. Also, they are less likely to be removed from the systems, and on average almost 69% of them reached to the final release. Moreover, on average nearly 67% of total alive genealogies did not contain any line additions or deletions or identifier renaming. Since we have studied a variety of systems, the results also indicate that it is possible that such a trend holds even when the systems are implemented in different languages, are from different areas and are of different sizes. We have noticed that clones are perhaps more manageable in smaller systems compared to larger ones. In addition to continuing our empirical study with very large (e.g., for Linux Kernel releases) systems and with systems of other interesting programming/scripting languages (e.g., Python), we plan to adapt our genealogy extractor to the NiCad [24] clone detection tool. NiCad can accurately detect near-miss clones [24, 25], even when statements are added, deleted or modified in the copied fragments, and thus will enable us to conduct a similar study for such near-miss clones as well.

Acknowledgements: The authors would like to thank the four anonymous reviewers for their valuable comments, suggestions, and corrections in improving the paper. This work is supported in part by the Natural Sciences and Engineering Research Council of Canada.

8. References

[1] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta, "Analyzing Cloning Evolution in the Linux Kernel", *Infor. & Soft. Tech.,* 44(13), 2002, pp. 755–765.

[2] L. Aversano, L. Cerulo, and M.D. Penta, "How Clones are Maintained: An Empirical Study", in *CSMR*, Amsterdam, 2007, pp. 81-90.

[3] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone Smells in Software Evolution", in *ICSM*, Paris, 2007, pp. 24–33.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, "Comparison and Evaluation of Clone Detection Tools", *IEEE Transactions on Soft. Eng.,* 33(9), 2007, pp. 577-591.

[5] 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", in *WCRE*, Lille, 2009, pp. 85-94.

[6] The CAROL: http://carol.ow2.org/ (March, 2010)

[7] The CCFinder: www.ccfinder.net (February, 2010)

[8] R. Geiger, B. Fluri, H.C. Gall, and M. Pinzger, "Relation of Code Clones and Change Couplings", in *FASE*, Vienna, 2006, pp. 411-425.

[9] N. Göde, "Evolution of Type-1 Clones", in *SCAM*, Edmonton, 2009, pp. 77-86.

[10] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do Code Clones Matter?", in *ICSE*, Vancouver, 2009, pp. 485-495.

[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 Soft. Eng.,* 28(7), 2002, pp. 654-670.

[12] C.J. Kapser, and M.W. Godfrey, ""Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software", *Emp. Soft. Eng.,* 13(6), 2008, pp. 645-692.

[13] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL", in *Sym. on Emp. Soft. Eng.*, Redondo Beach, 2004, pp. 83-92.

[14] M. Kim, and D. Notkin, "Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones", in *MSR*, Saint Louis, 2005, pp. 17-23.

[15] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies", in *FSE*, Lisbon, 2005, pp. 187-196.

[16] J. Krinke, "A Study of Consistent and Inconsistent Changes to Code Clones", in *WCRE*, Vancouver, 2007, pp. 170-178.

[17] J. Krinke, "Is Cloned Code More Stable Than Non-Cloned Code?", in *SCAM*, Beijing, 2008, pp. 57-66.

[18] B. Lagüe, D. Proulx, J. Mayrand, E. Merlo, and J.P. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process", in *ICSM*, Bari, 1997, pp. 314-321.

[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-Paste and Related Bugs in Operating System Code", in *OSDI*, San Francisco, 2004, pp. 289–302.

[20] A. Lozano, and M. Wermelinger, "Assessing the Effect of Clones on Changeability", in *ICSM*, Beijing, 2008, pp. 227-236.

[21] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the Harmfulness of Cloning: A Change Based Experiment", in *MSR*, Minneapolis, 2007, pp. 18-21.

[22] M. Rieger, S. Ducasse, and M. Lanza, "Insights into System–Wide Code Duplication", in *WCRE*, Delft University of Technology*,* 2004, pp. 100-109.

[23] C.K. Roy, and J.R. Cordy, *A Survey on Software Clone Detection Research,* Queen's School of Computing Tech. Report 2007-541, Kingston, 2007, 115 pp.

[24] C.K. Roy, and J.R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", in *ICPC,* Amsterdam, *2008,* pp. 172-181.

[25] C.K. Roy, and J.R. Cordy, "Near-miss Function Clones in Open Source Software: An Empirical Study", *Journal of Soft. Main. and Evolution,* 22(3), 2010, pp. 165-189.

[26] The Source Forge: http://sourceforge.net (March, 2010)

[27] S. Thummalapenta, L. Cerulo, L. Aversano, and M.Di Penta, "An Empirical Study on the Maintenance of Source Code Clones", *Emp. Soft. Eng.*, 15(1), 2010, pp. 1-34.