Analyzing and Forecasting Near-miss Clones in Evolving Software: An Empirical Study

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

***Abstract*—Effort for development and maintenance of complex large software is believed to have dependency on the amount of duplicated code fragments (code clones) present in code-bases. For example, clones need to be carefully and consistently maintained and/or refactored for preventing accidental error propagation. Thus it is important to understand the proportion and evolution of clones in evolving software systems for cost estimation or the like. This paper presents a study on the evolution of near-miss clones at release level in medium to large open source software systems of different types (operating systems, database systems, editors, etc.) written in three different programming languages namely C, C#, and Java. Using a hybrid clone detector, NiCad, we detected both exact and near-miss clones at different levels of similarity. Applying statistical methods we investigated, from different dimensions, the evolution of both exact and near-miss clones, and also forecasted the amount of clones in future releases of the software systems. Our study offers significant insights into the existence and evolution of code clones and their relationships with programming language or paradigm and program size.**

I. Introduction

Similar code fragments (functions, blocks, etc.) in software systems are known as code clones, and programmers' copy-paste-modification practice is regarded as one of the main reasons for majority of clones. Such clones are known as intentional clones. However, unintentional clones also appear due to a number of reasons. For example, the use of design patterns, frameworks, and similar API's result in unintentional code clones. Previous studies showed that software systems may have 5%-15% duplicated code [29], up to 50% [24]. Code fragments that are exact copies of each other form groups of *exact clones*. Moreover, fragments that are not exact copies of each other, but share certain level of similarity are known as *near-miss clones* where statements might be added/deleted/modified in the copied fragments. From the programmers point of view copy-paste-modification programming may increase productivity. However, copying a fragment containing any unknown bug may result in fault propagation. From the maintenance perspective, existence of code clones may increase maintenance effort. For example, a change in a cloned fragment may require careful and consistent changes to the all copies of the fragment. Any inconsistency may introduce new bugs. For small and simple software systems, such consequences may be minimal, but for complex and large systems, the consequences of code cloning may have significant impact in the development and maintenance process. Previous studies [3], [7], [11], [13], [20] on the usefulness/harmfulness of code clones converge to the point that code clones need to be carefully managed in evolving software. Thus, evolution and management of clones becomes an active research topic in the last few years.

Until recently, many studies on code clones reported observations on the effect of program size and programming language/paradigm on code cloning. However, to develop a more confident understanding on such phenomena, a larger scale structured study with statistical analysis on many *releases* of diverse systems is still required. Most of the other earlier works [8], [14], [16], [17], [21] on clone evolution investigated how individual cloned fragments evolve across subsequent CVS commit transactions or CVS snapshots over weekly intervals or so. Though such fine-grained studies offer important insights into the maintenance implications of code clones, a broader picture through analysis of clone densities at *release level* is also necessary. This will provide an overall but quick understanding of clones, which might help in estimating the maintenance cost of the software. Since the amount of cloned code is believed to have significant effect on software development and maintenance effort, project planning and management activities need deeper understanding on how much clones there may be in an evolving software. The ability to predict the amount of clones in the future releases would be useful in taking decisions on software cost estimation and project planning.

Most of the previous works studied the evolution of *Type-1* (identical code fragments except for variations in whitespace and comments) and/or *Type-2* (where syntactically similar fragments are also considered clones) clones. The works of Antoniol et al. [1], [2] might have included *Type-3* (statements added/deleted/modified in copied fragments) clones, but their metric based approach for clone detection might have included many false positives.

In this paper, we present an empirical study on the existence and evolution of exact and near-miss clones over 1,636 *releases* (and pre-releases) of 18 large open source software systems of diverse categories written in three different languages namely Java, C#, and C. To the best of our knowledge, this is the largest study so far on code clone evolution. Moreover, this study includes not only *Type-1* and *Type-2*, but also *Type-3* clones. Our study has two goals: first, using regression analysis we aim to compute one step ahead forecast on clone density in subsequent releases starting from a very early release of the system. Second, applying statistical methods we investigate

cloning property for understanding to what extent previous observations hold over a large number of releases of a wide variety of systems written in diverse programming languages. In particular, we focus on the following research questions: (a) Using a simple statistical model how accurately can we predict the amount of code clones in future releases? (b) Is there any common pattern in the evolution of clone density over releases of evolving software systems? (c) Is there any significant difference between the existence and evolution of exact clones and near-miss clones? (d) Do programming languages/paradigms have any effect on the amount and evolution of code clones in the evolving systems? and (e) How do the sizes of systems and functions affect density of exact and near-miss clones?

Through extensive quantitative analysis and manual investigation over 1,636 releases of the 18 software systems, we draw the following conclusions.

- Using simple regression analysis technique it is possible to make fairly accurate one step ahead forecast of clone density in future versions of software systems.
- Programming language/paradigm is found to have significant effect on code cloning. Java systems are found to have the highest amount of function clones, C systems have the lowest clone density, and the C# systems fit in the middle. Systems developed using object-oriented (OO) language/paradigm (Java and C#) have higher proportion of exact clones than near-miss clones, whereas the opposite holds for systems written using procedural C language. During evolution, Java and C# systems exhibit higher variation of clone densities than C systems. Moreover, there exists little or no effect of system's size on the regularity in the evolution of clone density.
- With the growth of software systems, as the number of functions increases, the number of both exact and near-miss cloned fragments also increases, indicating a very strong positive correlation between them. On the other hand, the correlation between clone density and number of functions is positive, but fairly weak, and larger systems tend to exhibit less clone density. As the similarity threshold decreases from near-miss clones to exact clones, the correlation between clone density and number of functions gradually gets weaker.
- The rate of change in clone density of near-miss clones is relatively irregular than that of exact clones in all the subject systems regardless of languages/paradigms and types of systems.
- There are some common patterns in the evolution of clone density across subsequent versions. For instance, relatively higher rate of changes in clone densities is found over early versions of software evolution. In the later releases, there exists long sequences of versions having relatively much less variations in clone densities.

The rest of the paper is organized as follows. In Section II, we provide the details of the experimental setup and then Section III presents the findings of our study. In Section IV we discuss the previous work on clone evolution and attempt to place our work in that context by providing a qualitative comparison between the studies. Section V shows the threats to the validity of our results and, finally Section VI concludes the paper with our directions of future research.

II. Experimental Setup

In this experiment we applied the NiCad clone detection tool [28] to find *function* level clones in release versions of a number of open source systems. Then we used a clone density metric, *Pearson's correlation coefficient* and regression analysis to analyze the results. This section introduces the studied systems and describes the methodology and metrics used, including a brief overview of our approach for manual verification of the detected clones.

A. Clone Density and its Correlation Coefficient

We conducted our analysis using the notion of clone density, which is the percentage of cloned functions over all functions in the system. Mathematically,

$$clone\ density = \frac{f_c \times 100}{f_c + f_{nc}} \tag{1}$$

where, f_c denotes the number of cloned functions, and f_{nc} refers to the number of non-cloned functions. Although we studied a similar metric w.r.t. lines of code, we omitted this in presenting the results due to both space limitation and strong correlation between these two metrics [25].

According to the above definition, having the number of cloned functions unchanged, if the number of total functions increases, clone density is expected to decrease in the subsequent releases. Intuitively, an increase in the total number of functions also increases the chance of more clones. Hence, the investigation of changes in clone density necessitates understanding the change relationship among total number of functions, number of cloned functions, and clone density. To examine this, we used *Pearson product-moment correlation coefficient* [22], which is a well-established statistical measurement to examine linear relationship between variables. We chose to use Pearson coefficient because it is suitable for interval data. The *Pearson product-moment correlation coefficient* r_{xy} between variables x and y is calculated by:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{2}$$

where x_i and y_i are values of the variables x and y, n is the number of samples (values) available for those variables, $\bar{x}$ and $\bar{y}$ are respectively the mean of all n values of x and y.

The value of r_{xy} ranges between $+1.0$ and -1.0 and indicates to what extent the variables are positively or negatively correlated. Two variables are positively correlated if one increases, then the other also increases. Negative correlation between variables implies if one gets larger, then the other gets smaller. Positive value of r_{xy} implies positive correlation and negative value implies negative correlation. The closer r_{xy} to ± 1.0 the stronger the correlation relationship is. A value

of r_{xy} close to zero indicates very weak or no correlation between the variables.

For exact and near-miss clones of each system, we computed *Pearson coefficient* r_{fc} between the number of functions and the number of cloned functions across all versions. Similarly, we computed *Pearson coefficient* r_{fd} between the number of functions and clone density. These two measurements help understand the change of clone density. Positive correlation between the number of functions and the number of cloned functions indicates that as new functions are created, many of them are possibly created by copy-paste operations, or those functions are accidental clones due to incorporation of similar product feature, or tackling similar problems. Positive correlation between the number of functions and the density of clones implies the amount of new clones is much higher than the amount of newly created functions.

B. Forecasting Clone Density

We applied regression analysis [22] for forecasting clone density in the subsequent versions of the each software system. Using regression analysis we estimated one step ahead clone density in the subsequent versions starting from the very third version of the underlying software system. The model gradually adapts the trend as more data becomes available, and thus error of estimation is expected to be reduced in the forecasts for the subsequent versions.

For each software system in our study, we plotted the clone densities against subsequent versions both for determining the suitable regression function and for identifying the components of the time series. As expected, we did not find any seasonality or cyclic component, but we found randomness and linear trend. Hence, we decided on linear regression model with single predictor variable, written as,

$$\hat{y_i} = \hat{\alpha_i} + \hat{\beta_i} x_i$$

where $\hat{y_i}$ is the response (estimation), x_i is the predictor variable (sequence number of a version), $\hat{\alpha_i}$ and $\hat{\beta_i}$ are regression coefficients at the i^{th} estimation step. At the i^{th} estimation step the values of α_i and β_i are estimated using the following equations:

$$\hat{\alpha_{i+1}} = \bar{y_i} - \hat{\beta_{i+1}} \bar{x_i}$$

$$\hat{\beta_{i+1}} = \frac{\sum_{k=1}^{i}(x_k - \bar{x_k})(y_k - \bar{y_k})}{\sum_{k=1}^{i}(x_k - \bar{x_k})^2}$$

$$\bar{x_k} = \frac{\sum_{j=1}^{k} x_j}{k}$$

$$\bar{y_k} = \frac{\sum_{j=1}^{k} y_j}{k}$$

where y_i is the actual value (clone density) observed at step i. The hat ($\hat{}$) symbol over a variable indicates estimated value. At the i^{th} step we compute the standard error of estimation σ_i using the following formula:

$$\sigma_i = \sqrt{\frac{\sum_{k=1}^{i}(y_k - \hat{y_k})^2}{i}}. \quad (3)$$

And for a given system s, the average standard error of estimate ξ_s is given by:

$$\xi_s = \frac{\sum_{i \in V_s} \sigma_i}{|V_s|} \quad (4)$$

where V_s is the set of all versions of the system s, for which forecasts were made. The regression model tries to find the best fit of the clone density evolution patterns in the software systems. Hence, forecast errors are expected to be relatively low for those systems having very regular clone evolution patterns. Thus the magnitudes of the forecast errors over subsequent versions indicate how irregular the underlying evolution pattern is.

C. Subject Systems

Table I describes the 18 software systems, their average sizes, and number of releases we used as subjects of our study. The number of lines presented in the Table I excludes blank lines and commented lines. Furthermore, we consider .c, .java and .cs files only. Our choice of subject systems was based on three criteria: (a) availability of many release versions over long life span of the systems, (b) size of the systems to be sufficiently large for clone analysis, and (c) diversity in types (i.e., OS, editor, database, etc.) of systems to minimize domain effect on the results.

D. Clone detection

We used the NiCad clone detector [28] for detecting near-miss clones. For consistency with our earlier studies [25], [27], we considered all non-empty functions of at least 3 lines in pretty-printed format. We then use size-sensitive UPI (Unique Percentage of Items) thresholds [28] to find exact and near-miss function clones. For example, if the UPI threshold (UPIT) is 0%, we detect only exact clones; if the UPIT is 10%, we detect two functions as clones if at least 90% of the pretty-printed text lines are the same (i.e., if they are at most 10% different). In this study, we used the representative set of UPITs 0%, 10%, 20% and 30%, corresponding to editing changes of from 0% to 30%, or 0 to 3 lines in every ten. It was not possible to manually validate all the detected clones in all releases of all the systems. We validated the clones of three releases (the first and the last releases and another randomly selected release) for each system and experienced almost no false positives. We used NiCad's interactive HTML output to obtain an overall view of the original source of the clone classes. Then, we carried out pairwise comparisons on the original source code of the functions in each clone class using Linux *diff*, followed by manual examination with a greater difference limit than the UPIT. Manual validation of precision using this method is both easy and efficient [25], but measuring recall over all the functions was not possible due to the difficulty level of the approach. However, NiCad, as a clone detector, was reported to have high recall [26] as well as precision [25], [28].

TABLE I: Subject systems for the case study

Lang.	Systems	Types/domains of systems	Releases		Avg. LOCs per release	Functions (Avg.)	
			Ranges	Total		Total	Sizes
Java	Apache-Ant	Build tool	1.1 to 1.8.0.1	20	80544	7246	11
	ArgoUML	Modeling tool	0.8.1 to 0.30.1-beta	145	136737	10048	14
	Commander4j	ERP system	2.2 to 2.97	37	45164	2373	19
	DavMail	Email client	1.4.0 to 3.6.5-1000	20	8324	469	18
	JasperReports	Reporting tool	0.x.x to 3.7.2	70	81853	5848	14
	JEdit	Editor	30-pre-4 to 43-pre-18	66	74261	4131	18
	Over all six Java systems			358	71147	5019	16
C	Conky	System monitor for X based on torsmo	1.1 to 1.8.0	70	19987	485	41
	GCC	Compiler	1.21 to 4.5.0	79	915438	16661	55
	GIT	Version control system	0.01 to 1.7.0.5	154	89896	2121	42
	Linux kernel	Operating System	0.01 to 2.6.0-test11	459	823021	19359	43
	PostGreSQL	Database	1.08 to 9.0.6	157	391628	9070	43
	Samba	File and print server	1.6.07 to 3.5.2	180	294227	9016	33
	Over all six C systems			1099	422366	9452	43
C#	NANT	Build tool	0.1.3 to 0.90.1	22	27850	1026	27
	CruiseControl	Continuous integration server	0.7 to 1.5.6804.1	24	59905	3545	17
	iTextSharp	Editor	0.01 to 5.0.1.1	39	131657	5691	23
	ProcessHacker	Process viewer and memory editor	1.0.0.0 to 1.3.9.0	35	63385	473	134
	WixEdit	Editor	0.1.1 to 0.7.3	32	10999	556	20
	ZedGraph	Drawing library	1.1 to 5.0.4	27	37702	505	75
	Over all six C# systems			179	55250	1966	49
Total number of release and pre-release versions across all systems is 1,636							

III. Experimental Results

For every release of each of the 18 systems, we computed density of clones (using equation 1) at different UPITs. Then for each system, at each different UPIT, we computed the average density of clones over all releases, as presented in Table II. Over all releases of each system, we computed the *Pearson correlation coefficient* between number of functions and clones (r_{fc}), as well as between number of functions and clone-density (r_{fd}), as shown in Table IV. Applying the regression analysis model, we made one step ahead forecast on clone density in subsequent releases of the systems. For each release of a system we calculated the total number of functions, the number of cloned functions, the actual and forecasted clone densities.

We compared each forecasted clone density with the actual density and computed the standard error of estimate using Equation 3. For each of four different UPITs, using Equation 4, we calculated the average standard error of estimate over all releases of a system. The regression analysis model enabled us to obtain one step ahead forecast on clone densities with reasonable level of accuracy. The standard error of estimation averaged over all four levels of UPITs and all the systems is 2.35. Table V presents the average forecast errors at different UPITs for all the 18 systems.

In Figure 1, we present the density of exact and near-miss clones averaged over all systems categorized by programming languages. We found that for all of the four UPITs Java systems have the highest clone density, C systems have the lowest clone density, and C# systems fall in between. This finding is consistent with that reported by Roy and Cordy [25], where they mentioned that the existence of accessor methods/functions in Java and C# systems might be a possible reason for this difference. Moreover, average function size in Java systems in our study is much lower than those in C and C# systems (Table I). The effect of programming paradigm (OO versus procedural) might have caused such differences. Intuitively, the smaller the sizes of the functions, the higher the the statistical probability of their being clones. This may

TABLE II: Average clone density in each system

Lang.	Systems	Densities for UPI threshold 30%	20%	10%	0%
Java	Apache-Ant	20.43	17.49	15.99	15.73
	ArgoUML	26.79	21.59	16.85	15.83
	Commander4j	43.76	38.33	33.98	32.63
	DavMail	13.74	6.93	4.42	3.64
	JasperReports	40.39	38.18	35.32	33.72
	JEdit	13.57	9.20	5.98	5.69
	Average	26.45	21.95	18.76	17.87
C	Conky	9.04	5.96	2.63	1.98
	GCC	18.61	13.93	9.31	6.77
	GIT	2.92	1.74	1.03	0.84
	Linux Kernel	14.15	9.19	4.85	3.14
	PostGreSQL	13.01	7.27	3.03	1.93
	Samba	15.39	8.84	3.28	1.78
	Average	12.19	7.82	4.02	2.74
C#	NANT	16.73	12.63	8.69	8.42
	CruiseControl	10.87	6.80	4.46	4.38
	iTextSharp	18.43	15.05	12.59	11.76
	ProcessHacker	7.32	3.20	2.17	2.17
	WixEdit	15.44	12.11	7.65	6.93
	ZedGraph	17.19	13.89	9.67	9.33
	Average	14.33	10.61	7.54	7.17
Average		17.65	13.46	10.11	9.26

TABLE III: Clone density categorized by system-size

Sizes (LOC/release)	Densities for UPI threshold 30%	20%	10%	0%
Below 50K	19.32	14.97	11.17	10.49
Between 50K and 100K	15.92	12.77	10.83	10.42
Above 100K	17.73	12.64	8.32	6.87

TABLE IV: Pearson coefficients r_{fc} and r_{fd}

Lang.	Systems	r_{fc} 30%	r_{fc} 0%	r_{fd} 30%	r_{fd} 0%
Java	Apache-Ant	0.999	0.997	0.75	0.53
	ArgoUML	0.81	0.63	-0.22	-0.15
	Commander4j	0.998	0.99	0.67	0.50
	DavMail	0.98	0.97	0.88	0.92
	JasperReports	0.999	0.998	0.87	0.87
	JEdit	0.95	0.89	-0.55	0.42
C	Conky	0.89	0.12	0.47	-0.24
	GCC	0.99	0.99	0.94	0.91
	GIT	0.80	0.48	-0.76	-0.59
	Linux Kernel	0.995	0.99	0.69	0.68
	PostGreSQL	0.98	0.83	0.78	0.03
	Samba	0.97	0.91	0.89	0.16
C#	NANT	0.996	0.99	0.81	0.62
	CruiseControl.NET	0.84	0.23	0.02	-0.39
	iTextSharp	0.997	0.99	0.83	0.81
	ProcessHacker	0.997	0.99	-0.45	0.84
	WixEdit	0.94	0.88	0.38	0.39
	ZedGraph	0.92	0.84	0.92	0.82

TABLE V: Average forecast errors ξ_s

Lang.	Systems	ξ_s for UPI threshold 30%	20%	10%	0%	Avg. ξ
Java	Apache-Ant	2.61	2.27	2.20	2.10	2.29
	ArgoUML	5.01	5.30	5.38	5.24	5.23
	Commander4j	0.44	0.46	0.60	0.56	0.52
	DavMail	2.23	2.00	1.25	1.20	1.67
	JasperReports	6.41	6.48	6.53	6.49	6.48
	JEdit	0.43	0.44	0.38	0.37	0.41
	Average	2.86	2.82	2.72	2.66	2.77
C	Conky	2.18	1.83	1.27	1.20	1.62
	GCC	1.93	1.64	1.43	1.09	1.52
	GIT	1.44	1.31	1.05	1.07	1.22
	Linux Kernel	0.55	0.85	0.62	0.46	0.62
	PostGreSQL	0.92	0.87	0.78	0.69	0.82
	Samba	2.89	2.72	2.32	2.23	2.54
	Average	1.65	1.54	1.25	1.12	1.39
C#	NANT	4.05	4.07	3.56	3.60	3.82
	CruiseControl	2.29	2.10	1.77	1.76	1.98
	iTextSharp	4.55	4.62	4.01	3.74	4.23
	ProcessHacker	0.09	0.15	0.15	0.15	0.13
	WixEdit	4.48	4.76	4.36	3.73	4.34
	ZedGraph	3.32	3.13	2.68	2.61	2.93
	Average	3.13	3.14	2.76	2.60	2.91
Overall		2.55	2.50	2.24	2.13	2.35

be another reason for comparatively higher clone density in Java systems in our study. As the UPIT increases from 0% to 10%, 20%, and 30%, the density of clones largely increases in C systems. On the contrary, exact clones dominate in Java and C# systems, as with the increase of UPIT clone density does not increase much for these systems.

Figure 2 plots the actual and forecasted density of clones in all 459 releases of Linux kernel (the system in our study having the highest number of releases). The Java system JasperReports contributed the most in the standard error of estimate mentioned above. For this system, the average standard error of estimate is the highest (6.48), which also indicates high irregularities in its clone evolution, as shown in Figure 3. To keep the figure legible, we plotted the actual and forecasted density of clones only for exact clones and near-miss clones with threshold 30%. The *Pearson coefficients* r_{fc} and r_{fd} (Table IV) indicates that over subsequent releases of Jasper-

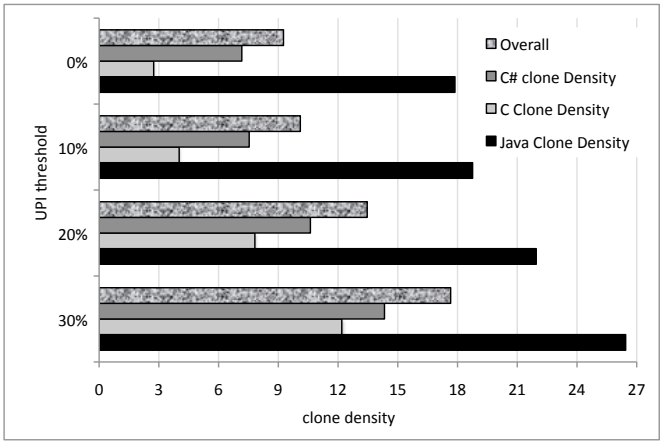

Fig. 1: Average density of exact and near-miss clones

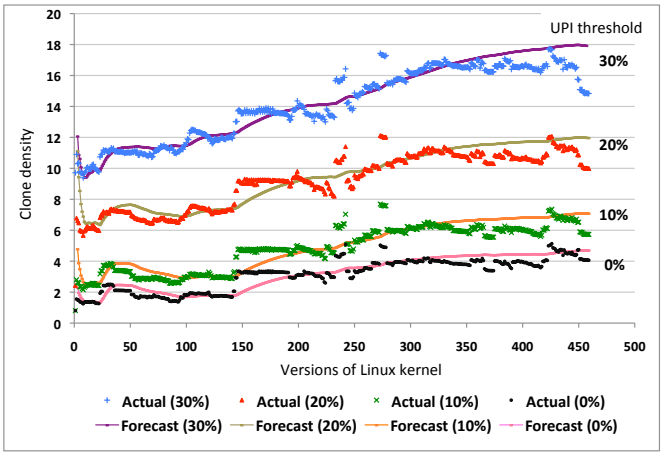

Fig. 2: Actual and forecasted density of exact and near-miss clones in Linux kernel

Reports as the number of functions increased or decreased, the number of clones as well as clone-density also changed in the same direction. This suggests that the fluctuations in JasperReports' clone density is due to major changes in the number of functions in the subsequent releases. This is further validated by the data for individual releases. For example, we found that between JasperReports 0.2.5 and JasperReports

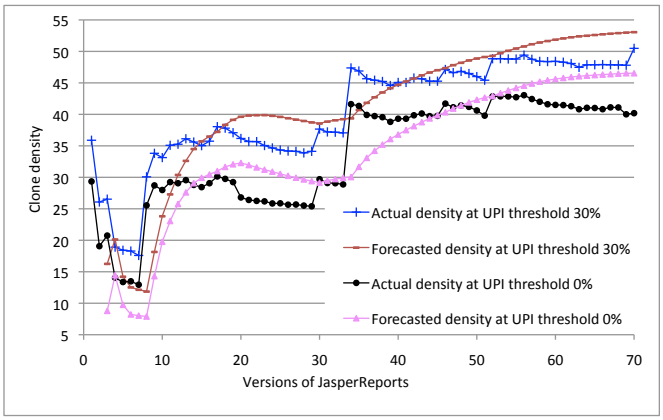

Fig. 3: Actual and forecasted density of exact and near-miss clones in JasperReports

0.3.1 the number of functions increased from 796 to 1,147 and the number of exact cloned functions increased from 103 to 293 resulting increase in clone density from 12.9% to 25.5%. Looking at the change log for JasperReports 0.3.1, we found that besides bug fixes and architectural improvements from earlier release, support for XML output was added in release 0.3.1, which may be a reason for such large increase in number of functions and clones.

As we see in Table V, the forecast errors averaged over C systems are much lower (1.39) than that of Java and C# systems. For Java and C# systems the average forecast errors (2.77 and 2.91 respectively), are more than twice as for C systems. This implies that the clone evolution patterns in Java and C# systems are more irregular than in C systems. However, among Java systems JasperReports and ArgoUML produced the highest average forecast errors, 6.48 and 5.23 respectively. On the other hand, WixEdit and NANT among all C# systems produced the most forecast errors, 4.34 and 3.82 respectively.

As the UPIT increases from 0% to 30%, for most of the systems across all three languages the average forecast errors (Table V) also increase indicating that the evolution of near-miss clones is more irregular than that of exact clones. The reason may be the fact that any two fragments have higher chance of being similar than being exactly same, and so, creation or deletion of a functions may have higher effect on the amount of near-miss clones than exact clones. Possibly, due to the same reason, for almost all systems, as UPIT increases from 0% to 30%, the *Pearson coefficient* r_{fc} (between number of functions and number of cloned functions) gets closer to $+1.0$ (Table IV). We see that for all systems, r_{fc} is positive for both exact and near-miss clones. For all systems r_{fc} is +0.80 or more for near-miss clones (UPIT 30%), which suggests a very strong positive correlation between the number of functions and the number of near-miss cloned functions. In case of exact clones, r_{fc} is +0.5 or higher for 15 out of 18 systems, which also suggests fairly strong positive correlation.

On the other hand, the value of r_{fd} (correlation coefficient between the number of functions and clone density) is +0.5 or higher for 11 out of 18 systems at UPIT 30%, and for four systems r_{fd} is negative. This may be interpreted with the fact that for the near-miss clones at UPIT 30%, there exists weak positive correlation between the number of functions and clone density. For exact clones half of the systems exhibit r_{fd} values less than +0.5, and four of the systems has negative r_{fd}. This shows relatively weaker positive correlation between number of functions and density of exact clones. We also see that with the decrease in UPIT from 30% to 0%, positive correlation between number of functions and clone density gets weaker. However, looking at average clone densities categorized by systems' size (Table III) we see that for UPIT 0%, 10%, and 20% average clone density decreases as the systems' size increases from below 50 KLOC to over 100 KLOC. This further weakens the positive correlation between number of functions and clone density. A possible explanation may be that large systems in our study tend to have functions of larger sizes, and larger functions have statistically less

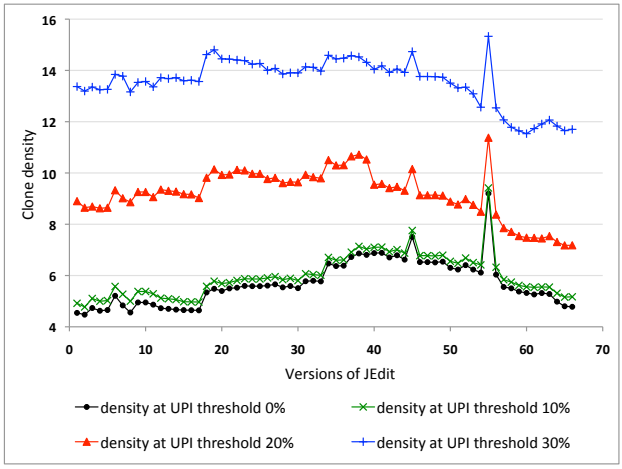

Fig. 4: Exact and near-miss clone density in JEdit

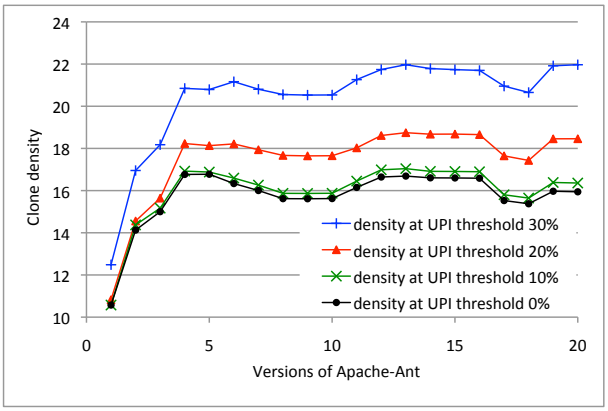

Fig. 5: Exact and near-miss clone density in Apache-Ant

probability of being clones than smaller functions. In our study the four largest systems are C systems namely GCC, Linux kernel, PostGreSQL, and Samba having average function size of 43 LOC, much higher than average Java function size (16 LOC). For instance, the "void ffebld_constantarray_prepare (...)" function in "gcc/f/bld.c" source file of GCC-3.0.1 has 361 pretty-printed LOC consisting of long sequences of switch statements.

Looking at the clone densities over all the systems across subsequent releases, we found interesting patterns in the evolution of clone density. For each system, clone densities for the four UPITs exhibit the same evolution pattern varying only in their magnitudes. For example, Figure 2 plots clone densities at UPITs 0%, 10%, 20%, and 30% with roughly parallel lines. We also observed that for most of the systems, the density of exact clones does not differ much from the density of clones at UPIT 10%. In Apache-Ant, NANT, JEdit, ArgoUML, GIT, GCC, ProcessHacker, WixEdit, CruiseControl.NET, and ZedGraph clone densities at UPIT 10% is found to be almost equal to the densities of exact clones. For other systems, we found differences, which is much lower than the differences of clone densities between UPITs 10% and 20%, or between threshold 20% and 30%.

Having some short term fluctuations of increase and decrease, over long term clone density tend to increase linearly

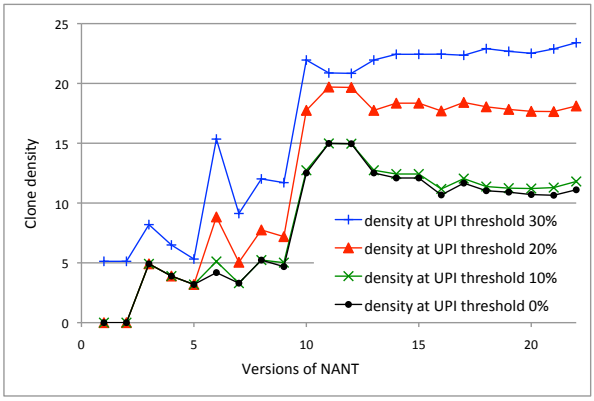

Fig. 6: Exact and near-miss clone density in NANT

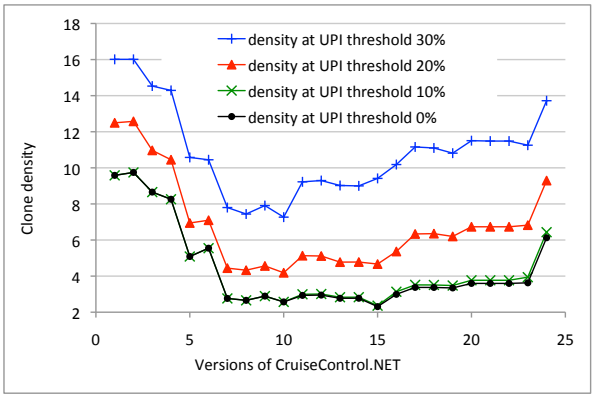

Fig. 7: Exact and near-miss clone density in CruiseControl.NET

in Linux kernel (Figure 2). Previous studies on Linux kernel reported roughly linear (super linear) growth of Linux kernel source code in terms of LOC [10]. The linear growth of both line of source code as well as density of code clones indicate that cloning activities continued in a similar pace as the development and maintenance activities took place in Linux kernel code-base. Similar geometric evolution of clone density is found in JEdit. For this system, at all four UPITs clone density gradually increased over the early 38 releases between JEdit-3.0-pre4 and JEdit-4.2-pre2. Then over the later 28 releases between JEdit-4.2-pre3 and JEdit-4.3-pre18 clone density decreased. However, we found roughly linear growth in the number of functions over all releases of JEdit. This suggests possibly more copy-paste-modification activities took place during the early development of JEdit due to active development of similar features implementation, and as the project became stable, later development activities were possibly dominated by maintenance, and performance enhancement rather than new feature incorporation.

For all the other 16 systems (except Linux kernel and JEdit), we found some interesting patterns in the evolution of clone densities. In all these 16 systems, we noticed major change in clone densities over early releases, and relatively smaller variations over sequence of many later releases. A reason to this fact may be during the early releases active development and feature implementations dominated in software growth,

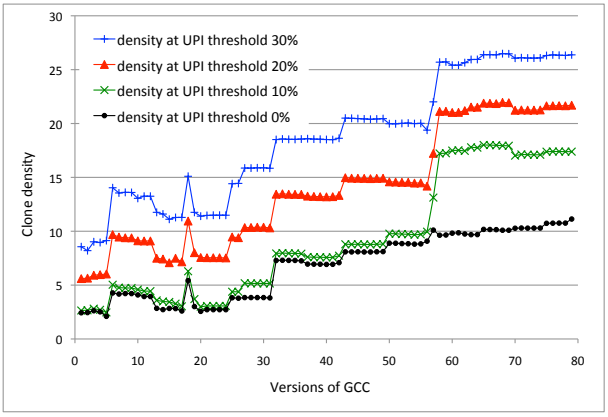

Fig. 8: Exact and near-miss clone density in GCC

whereas maintenance activities dominated in the later releases. As shown in Figure 5, clone density strictly increased in Apache-Ant from release 1.1 (10.6% exact clones) to release 1.4 (16.8% exact clones) and did not vary much (amount of exact clones remained between 15.38% 16.8%) over the later 16 releases. Such pattern was also found in Commander4j. Similar pattern of increasing clone density was also found in DavMail, Conky, NANT, and WixEdit. However, in these systems, the early change in not strictly increasing, rather overall increase with short term fluctuations is found. For example, clone density in NANT (Figure 6) increased from release 0.1.3 (5.1% near-miss clones at UPIT 30%) to 0.8.3 (20.9% near-miss clones at UPIT 30%), and did not exhibit much variations over the later releases (density of near-miss clones at UPIT 30% remained between 20.9% and 23.4%). However, between NANT-0.1.3 and NANT-0.8.3, there exists some fluctuations of increase and decrease in clone density. From NANT-0.1.4 clone density increased in NANT-0.1.5, again decreased through release 0.5 and 0.6, which followed an increased density in NANT-0.7.7. This is possibly due to active development and refactoring activities during those early releases of NANT.

A second pattern for clone evolution during early developments was found in CruiseControl.NET, ArgoUML, GIT, JasperReports, PostGreSQL, and ZedGraph. For all these systems, during the early releases, clone densities tend to decrease along with some short term fluctuations. Figure 7 shows this pattern plotting the evolution of clone density for CruiseControl.NET. For this system, as we see, clone density largely decreased from release 0.7 (9.6% exact clones) through release 1.0 (2.8% exact clones), and over later 16 releases (release 1.0.1 through release 1.5.0) density of exact clones remained between 2.3% and 3.6%. However, we found that over those releases of CruiseControl.NET, the number of functions increased from 2,473 to 2,859, which indicates the possibility of not much copy-paste-modification activities took place during the early developments of CruiseControl.NET.

After the early phase of software evolution, over long term relatively less variation in clone density was found in general. However, two interesting patterns are found. The first pattern exhibits roughly steady clone density over long sequence

of releases with small irregular variations. Such pattern is found in WixEdit, iTextSharp, DavMail, Commander4j, Ant (Figure 5), NANT (Figure 6), CruiseControl.NET (Figure 7), and JasperReports (Figure 3). In the second pattern, clone density is found not to vary that much over long sequence of releases, and relatively large changes found between such release sequences. Similar pattern is found in GCC, ArgoUML, GIT, JasperReports, PostGreSQL, Samba, and ProcessHacker. Scatter plot of such pattern yields a stair-like shape, as shown in Figure 8, which shows exact and near-miss clone densities in 79 releases of GCC. There are two possibilities to cause such steady clone density: when neither the number of functions nor the number of cloned fragments changes, or when both changes in a ratio such that clone density remains steady. Interestingly, both of these facts are found in systems exhibiting such patterns. For example, density of exact clones remained between 3.78% and 3.84% over seven releases of GCC, GCC-2.8.0 through GCC-2.95.3. Among these releases, between GCC-2.8.1 and GCC-2.95 the number of functions increased from 5,893 to 11,268, and the number of cloned functions also increased from 223 to 433 causing no significant change in clone density. Again, from GCC-2.95.3 to GCC-3.0 the number of functions increased from 11,300 to 15,936, and number of cloned functions increased from 431 to 1161, causing clone density jump from 3.81% to 7.28%. Over the following four releases, GCC-3.0.1 through GCC-3.0.4 the total number of functions varied a bit between 16,016 and 16,134. The number of cloned functions remained unchanged at 1,170 and consequently the clone density varied a little between 7.25% and 7.31%.

Besides the above mentioned patterns in the evolution of clone density in the later releases, some systems exhibit sudden significant increase in clone densities between subsequent releases followed by large decrease in the next release. Such sudden variations are found in JEdit, ArgoUML, GIT, ZedGraph, and Samba. For instance, the large spike in Figure 4 corresponds to JEdit-4.3-pre6 between JEdit-4.3-pre5 and JEdit-4.3-pre7. In JEdit-4.3-pre6 the number of clones increased significantly compared to its previous and later releases.

In our earlier work [31] we tracked individual clone groups (genealogies and lineages [14]) over releases and characterized them in terms of dead, alive, consistently changed, and syntactic similar genealogies. However, the length of the succession of individual clone groups over releases is yet to be investigated to understand how persistent the clone-groups are across releases. A clone-group in a later release is said to be a successor of a clone-group of an earlier release, if the code snippets in the earlier release does not change more than a threshold (30% lines in our study) in the later release. The length of succession of a clone-group is the maximum number of subsequent releases in which a successor of the clone-group exists.

For the two Java systems DavMail and Apache-Ant, we tracked individual clone groups, and found that a significant amount of clone-groups have succession length over 60% of

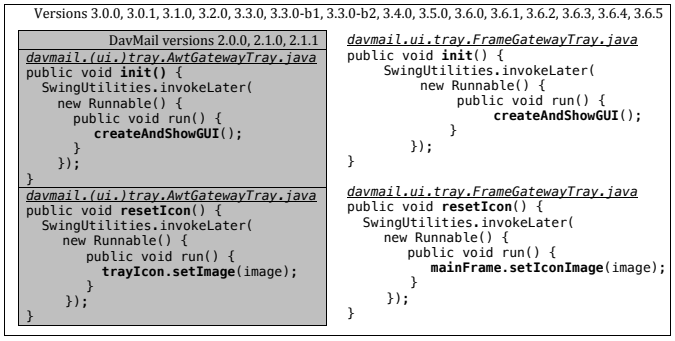

Fig. 9: Group of near-miss function clones

the total number of releases. In DavMail 38% of the clone-group successions have length 80% of the releases or more. In Figure 9 we present an example of such succession of near-miss clones. The clone-group consisting of the two shaded functions in the figure have succession spanned over releases 2.0.0 through 2.1.1. The other two functions joined the clone-group in the release 3.0.0, and the succession continued to release 3.6.5 resulting a total succession length of 18 releases out of 20. Due to space limitation, a detail analysis on clone-group succession is kept aside as a separate study in the future.

IV. Related Works

Studying the characteristics and evolution of code clones is not a new topic and there have been several studies in the past. While they differ significantly in many aspects, they are also related to our study.

Roy and Cordy [25] conducted an empirical study on 23 systems written in three different languages, and examined distribution of clones in the systems as well as the effect of programming language and system's size. Later they conducted another study [30] on eight systems written in Python, and found that cloning properties of Python systems are not really different from previous observations for C, Java, and C#. However, both the studies examined code clones in single version of each system, without investigating clone evolution across software versions.

Laguë 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 because we not only study clone densities but also forecast clone densities for the future releases of the software and that we conduct the study with 18 diverse categories of systems of three different languages and we use a hybrid clone detection method [28], which is has high precision [25] and recall [26].

Göde [8] examined the evolution of individual exact cloned fragments in several open source systems written in C, C++, and Java using iClones [9]. The ratio of exact clones over system size decreased in majority of the systems he studied.

Krinke [16] on the other hand, conducted an empirical study to investigate the type of changes taking place on cloned versus non-cloned codes to determine whether cloned code is more stable than non-cloned code. For clone detection, he used Simian, a text-based clone detector capable of identifying almost identical clones. He observed that changes to the evolving software is dominated by massive deletion of cloned codes, and if deletion is ignored, in terms of line addition and modification, cloned code is more stable than non-cloned code. Krinke [17] also 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.

A number of tools and frameworks [6], [23] have been developed for aiding clone tracking, analysis, evolution and management. Kim et al. [14], [15] coined the terms "clone lineage" and "clone genealogy" to describe relationship between clone groups in subsequent versions of evolving software. To investigate the clone evolution, they developed a clone genealogy extractor using CCFinder [12], a token based *Type-1* and *Type-2* clone detector. To evaluate their approach, they extracted clone genealogy from two open-source Java projects, Carol and DnsJava. They found that many genealogies disappear in a relatively short time after their birth. Lozano and Wermelinger [21] analyzed commit-by-commit evolution of five open source Java projects over at least thirty months. They also used CCFinder [12] for detecting clones in methods. They found that cloned methods change more than non-cloned method, which contradicts Krinke's findings [16]. They further reported that cloned methods tend to remain cloned most of their lifetime, which also contradicts the findings of Kim and Notkin [15]. 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. Thummalapenta et al. [33] proposed an automatic approach for classifying the evolution of cloned fragments and reported an analysis using four different Java and C software systems for investigating to what extent clones are consistently propagated or they evolve independently. Bettenburg et al. [4] studied the inconsistent changes of clones at the release level. They noted that the number of defects due to inconsistent changes in clones is substantially lower at the release level than at the revision level.

While these studies provide important insights on the fine-grained evolution of clones and their maintenance implications, our study significantly differs from them in several aspects. First, instead of providing in-depth analysis on the evolution of clones, we provide an overall analysis on the evolution of clone densities and attempt to forecast such densities in the future releases for 18 diverse varieties of systems of three different programming languages. None of the studies above attempted to forecast clone densities as ours. Second, a common issue with most of these studies is that they are based on CVS snapshots over certain intervals. The interval of commit-by-commit transactions or snapshots of CVS/SVN repository over certain periods may be too frequent for analyzing clone evolution [4]. Moreover, commit transactions are sensitive to developer's commit style. So, we studied at the release (and pre-release) level. Finally, most of the earlier works studied the evolution of *Type-1* and/or *Type-2* clones, whereas we studied both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones at different similarity levels using a hybrid clone detection tool.

Our work closely relates to that of Antoniol et al. [2]. They applied the *ARIMA(p,d,q)* [5] time series analysis to model code clone evolution for predicting amount of code clones in 27 subsequent versions of MiniSQL (written in C) using a metrics-based clone detection tool. Again, our study significantly differs from them in examining 18 systems of diverse varieties written in three different programming languages, and using a parser-based but text-line comparison clone detection tool with different similarity levels. Moreover, the *ARIMA* process is elegant but complicated, and applicable only when there is enough historical data available to generate a reliable model. The very first step in the *ARIMA* process is to identify a tentative model through the analysis of historical data, and it is recommended that 50, preferably more historical observations be considered at this step [22]. Shawky et. al.[32] modelled clone evolution in two software (i.e., 50 versions of FileZilla and 100 versions of VLC) using *chaos theory* for prediction in new versions. *Chaos theory* also requires historical data to build the initial model. This implies that neither *ARIMA* nor the *chaos theory* is suitable for estimation during the early stage in the evolution, whereas the regression analysis technique is applicable from the very early releases of the software evolution.

V. Threats to Validity

In our study, we examined multiple releases of six systems for each of three programming languages (C, C#, and Java) which leads to a total of 18 subject systems. This number of systems for each language may not be enough to derive decisive conclusion on the effect of programming languages/-paradigms on code clone evolution. However, this is the largest study in clone evolution that considers a total of 1,636 releases over 18 subject systems of diverse varieties.

There exists the open question of what should be the appropriate level of similarity to consider two code fragments as near-miss clones. To address this concern, we examined evolution of code clones at four different levels of similarities: exact clones, and near-miss clones with 70%, 80%, and 90% similarities on the pretty-printed source lines of code (see [25], [28] for details.)

Since we considered all functions having at least three LOC in pretty-printed format, one might argue that the findings are biased on the size of the functions. However, in the pretty-printed format used in our study, the function header and opening brace constitute the first line, and the last line contains at least one line of code followed by ending brace(s). Thus we filter out those getter and setter functions that contain only one return or assignment statement in the function body. Moreover,

earlier work [25] showed that the proportions of clones in terms of LOC and the number of functions are very close.

Though we carried out some manual verifications, exhaustive manual validation over all the found clones for all the releases of all the systems was not possible. However, we used NiCad clone detection tool which was reported to have high precision [25], [28] and recall [26] in finding clones.

VI. Conclusion

In this paper, we presented an empirical study on the evolution of exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) code clones in 18 large open-source software of diverse categories written in three different languages namely Java, C#, and C. For each software system our study included many releases (and pre-release) ranging between 20 and 459, summing up to 1,636 in total. For each system we investigated how the density of code clones changes between subsequent releases at four different similarity levels. Using the *Pearson product moment correlation coefficient* we examined the relationship between software growth (in terms of number of functions) and changes in the amount of clones. We found that with the increase in the number of functions, the number of cloned fragments also increases in all systems. However, between clone density and the number of functions, very weak positive correlation has been identified. Moreover, the average clone density over larger (in terms of LOC) systems are found to be less than that over smaller systems. Using simple regression analysis we were able to make one step ahead forecast on the density of both exact and near-miss clones in subsequent releases. The average standard error of estimate over all releases of all systems was 2.35. We tracked the forecast errors over subsequent releases of each system and examined the regularity of changes in clone density from release to release.

We also identified some interesting patterns in the evolution of clone density over subsequent releases. For instance, we found major changes in clone density over few early releases of software life time, and over the later releases there exists long sequences of releases among which clone density does not vary that much. We plan to perform a separate in depth investigation on the relationship between clone density and function size. In addition to continuing our empirical study with very large systems and with systems of other programming/scripting languages (e.g., Python), we plan to develop a framework on top of NiCad clone detection tool that will track the evolution of clones, which might provide important insights into the maintenance implications of clones during the evolution of software systems.

References

[1] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the Linux kernel. *Infor. & Soft. Tech.* 44(13): 755–765, 2002.

[2] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *ICSM*, pp. 273–280, 2001.

[3] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *ICSM*, pp. 24–33, 2007.

[4] 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*, pp. 85–94, 2009.

[5] G. Box, E. Pelham, and G. Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[6] E. Duala-Ekoko, and M. P. Robillard. Tracking code clones in evolving software. In *ICSE*, pp. 158–167, 2007.

[7] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *FASE*, pp. 411–425, 2006.

[8] N. Göde. Evolution of Type-1 clones. In *SCAM*, pp. 77–86, 2009.

[9] N. Göde, and R. Koschke. Incremental clone detection. In *CSMR*, pp. 219–228, 2009.

[10] M. W. Godfrey, and Q. Tu. Evolution in open source software: A case study. In *ICSM*, pp. 131–142, 2000.

[11] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, pp. 485–495, 2009.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. In *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[13] C. J. Kapser, and M. W. Godfrey. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. In *Empirical. Software Engineering*. 13(6): 645–692, 2008.

[14] M. Kim, and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR*, pp. 17–23, 2005.

[15] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *FSE*, pp. 187–196, 2005.

[16] J. Krinke. Is cloned code more stable than non-cloned code? In *SCAM*, pp. 57–66, 2008.

[17] J. Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *WCRE*, pp. 170–178, 2007.

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

[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*, pp. 289–302, 2004.

[20] A. Lozano, and M. Wermelinger. Assessing the effect of clones on changeability. In *ICSM*, pp. 227–236, 2008.

[21] A. Lozano, and M. Wermelinger. Tracking clones' imprint. In *IWSC*, pp. 65–72, 2010.

[22] D. C. Montgomery, C. L. Jennings, and M. Kulahci. *Introduction to Time Series Analysis and ForeCasting*. Wiley Series in Probability and Statistics. John Wiley and Sons, Inc., 2008.

[23] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Cleman: Comprehensive clone group evolution management. In *ASE*, pp. 451–454, 2008.

[24] M. Rieger, S. Ducasse, and M. Lanza. Insights into System-Wide Code Duplication. In *WCRE*, pp. 100–109, 2004.

[25] C. K. Roy, and J. R. Cordy. Near-miss function clones in open source software: an empirical study. In *Journal of Soft. Maintenance and Evolution: Research and Practice*, 22(3):165–189, 2010.

[26] C. K. Roy and J. R. Cordy. A Mutation / Injection-based Automatic Framework for Evaluating Clone Detection Tools. In *Mutation'09*, pp. 157–166, 2009.

[27] C. K. Roy and J. R. Cordy. An Empirical Study of Function Clones in Open Source Software Systems. In *WCRE*, pp. 81–90, 2008.

[28] C. K. Roy, and J. R. Cordy. NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pp. 172–181, 2008.

[29] C. K. Roy and J. R. Cordy. *A survey on software clone detection research*. Queen's School of Computing Tech Report 2007-541, 115 pp., 2007.

[30] C. K. Roy and J. R. Cordy. Are Scripting Languages Really Different? In *IWSC*, pp. 17– 24, 2010.

[31] 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. In *SCAM*, pp. 87 – 96, 2010.

[32] D. M. Shawky and A. F. Ali. Modeling Clones Evolution in Open Source Systems Through Chaos Theory. In *ICSTE*, pp. V1-159 – V1-164, 2010.

[33] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. In *Empirical Software Engineering*, 15(1):1–34, 2010.