Analysis and Management of Code Clones

Minhaz Fahim Zibran
Department of Computer Science, University of Saskatchewan,
Saskatoon, SK, Canada S7N 5C9
Email: minhaz.zibran@usask.ca

***Abstract*—Copy-paste programming may speed-up software development process, but code clones, in the long run, might increase maintenance cost. Thus, despite the ongoing research on the usefulness/harmfulness of code clones, researchers and practitioners are in accord on the point that clones need to be managed. However, to devise efficient clone management techniques, one needs accurate and deep understanding on the existence and evolution of clones, as well as the factors that affect such phenomena. In this doctoral symposium paper, I present my ongoing research towards code clone analysis, management, and refactoring.**

I. Introduction and Motivation

Similar code fragments (functions, blocks, etc.) in software systems are known as code clones. To be regarded as clones, a pair of code snippets have to be exact duplicates (*Type-1* clone), or the fragments may not be identical but can be syntactically similar (*Type-2* clone), or there may be one or more statements added/modified/deleted beyond the syntactic similarity (*Type-3* clone). A set of code fragments, where any two are clones of each other, form a *clone-group*. Previous studies reported that software systems may have 9%-17% duplicated code [20], up to 50% [16].

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, the existence of code clones may increase maintenance effort. For example, a change in a clone 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. Earlier studies [1], [4], [6], [7], [13] on the usefulness/harmfulness of code clones converge to the point that code clones need to be carefully managed in evolving software. Thus, the analysis and management of clones becomes an active research topic in the last few years.

In this mid-career doctoral symposium paper, I first present two separate studies, one high level and the other fine-grained analysis on code clones (Section II). Both these studies emphasize the significance of my ongoing work (described in Section III) towards an IDE-integrated clone management and refactoring system, which is the ultimate goal of my research. Finally, Section IV summarizes the research contributions and my directions to future research.

II. Analysis of Code Clone and Evolution

Here I present two separate empirical studies. The purpose of the studies was to obtain in-depth understanding on the code cloning phenomenon. The first study [20] analyzed over-all cloning status in evolving software systems and examined the effects of influencing factors such as programming language/-paradigm and system-size on the existence and evolution of code clones. The second study [19] carried out a fine-grained analysis, which investigated how the individual clone fragments changed and evolved across subsequent releases of evolving software systems.

A. Study-1: High Level Analysis

1) Research Questions: The study focused 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 (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones? (d) Do programming languages/paradigms have any effect on the existence and evolution of code clones in the evolving systems? and (e) How do the sizes of systems and functions affect the density of exact and near-miss clones?

2) Methodology: The study was conducted on 1,636 releases of 18 open-source software systems written in C, C#, and Java. Using `NiCad` [17], we first detected function level exact and near-miss clones in every release of each of the subject systems. Then *clone density* was calculated using the following equation,

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

where, f_c denotes the number of clone functions, and f_{nc} refers to the number of non-clone functions. Applying *regression analysis* we forecasted one step ahead clone density in the subsequent releases (starting from the very third release) of the underlying software systems. The average *standard error of estimates* over releases of certain system indicated how irregular the underlying clone evolution pattern was. Using the *Pearson product-moment correlation coefficient* [15], we investigated the change relationships among the total number of functions, number of cloned functions, and clone density.

3) Findings:

- Using simple regression analysis, it was possible to make fairly accurate one step ahead forecast of clone density

in future versions of software systems.

- Programming language/paradigm was found to have significant effect on code cloning. Java systems were 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#) had higher proportion of exact clones than near-miss clones, whereas the opposite held for systems written using procedural C language. During evolution, Java and C# systems exhibited higher variation of clone densities than C systems. System's size had little or no effect on the regularity in the evolution of clone density.
- As the number of functions increased with the growth of the systems, the number of both exact and near-miss clone fragments also increased.
- Some common patterns were discovered in the evolution of clone density across subsequent releases. For instance, relatively higher rate of changes in clone densities was found over early releases of software evolution and long sequences of later releases were found to have relatively much less variations in clone densities.

B. Study-2: Low Level Analysis

In this study [19], we performed an in-depth investigation of code clone genealogies in evolving open source systems at the release level. A *clone genealogy* (Figure 1) consists of a set of clone lineages that originate from the same clone-group. 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.

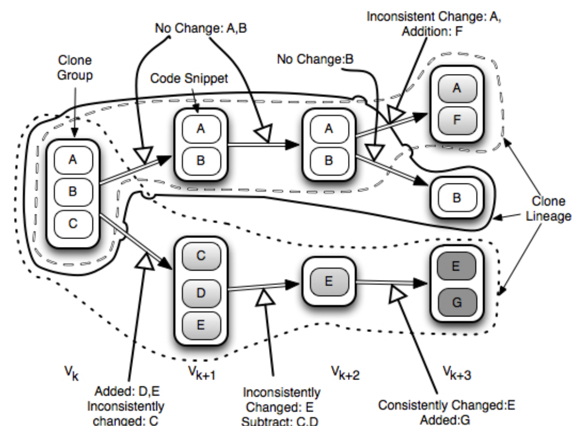

Fig. 1: Clone genealogy [19]

1) Research Questions: The study focused on two research questions: (a) How do the clone genealogies look like in open source software of different sizes, having variable release histories, and written in different languages? (b) 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?

2) Methodology: The study was conducted over 5,93 releases across 17 open-source diverse software systems written in four different languages namely C, C++, C#, and Java. Using `CCFinderX`[1], we detected *Type-1* and *Type-2* clones from each release of every subject system. Then for each of the 17 subject systems, we extracted clone genealogy using the *genealogy extractor* we developed. To analyze the change-patterns in evolving clone-groups, we characterized four types of clone genealogies. A genealogy was called *alive genealogy*, if it contained at least one clone-group carried up to the final release; otherwise, it was marked as a *dead genealogy*. *Syntactically similar genealogies* referred to those genealogies in which the clone-groups 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. *Consistently changed genealogies* were those genealogies where all the clone-groups had at least one consistent-change pattern of any sort (e.g., addition of a new line to all the snippets of the clone-groups). Further quantitative and qualitative analysis was performed on each type of clone genealogies to derive the findings.

3) Findings:

- Most of the clone-groups propagated through subsequent releases either without any changes, or with changes only in identifier names. Many of them reached to the final releases of the subject systems and contributed to the number of alive genealogies. On average over all the systems, about 67% of the genealogies did not have any addition or deletion of lines, or any syntactic changes. Moreover, on an average, roughly 69% of these syntactically similar genealogies reached to the final releases.
- About 11% to 38% of the genealogies changed consistently over the entire course of the evolution. Many of the dead genealogies disappeared within a few early releases.
- Programming languages or project-sizes did not have any significant effect on clone evolution.

Both the empirical studies revealed the existence of significant amount of code clones in software systems, which also evolved with the evolution of the systems themselves. Such clone evolution was often dictated by consistent and/or inconsistent changes in the clone-groups, as well as removal of clone fragments. This further intensifies the realized need for clone management with tool support.

III. Clone Management

Accurate detection of code clones is the fundamental and vital step towards clone management. Over the past decade several techniques and tools for detecting code clones have been proposed, having their own strengths and weaknesses [18]. While most of them are capable of detecting *Type-1* and *Type-2* clones, only a few of them are reported to detect *Type-3* clones. Indeed, it is not enough to only detect code clones. Code clones are required to be tracked, managed, and possibly should be removed through refactoring wherever feasible. And support for such activities should be integrated with the IDEs

[1]The clone detector was obtained from www.ccfinder.net

for blending clone management with actual development effort. However, most clone detectors are developed as separate tools. Those few tools that are integrated with IDEs are mostly focussed in detecting *Type-1* and *Type-2* clones, and are yet to offer sufficient support for flexible clone management and refactoring. Addressing these issues, we have been developing an IDE-based clone management system [21] for accurate and flexible detection, management, and refactoring of both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) code clones.

A. Clone Detection

We developed a language independent matching engine (LIME), a tool for fast localization of all k-difference (edit distance) occurrences of one code fragment inside another [21]. On top of LIME, we have developed a near-miss clone detection tool as a plugin to the Eclipse IDE. Figure 2 presents a schematic diagram of the major algorithmic modules and the process of clone detection used in our approach.

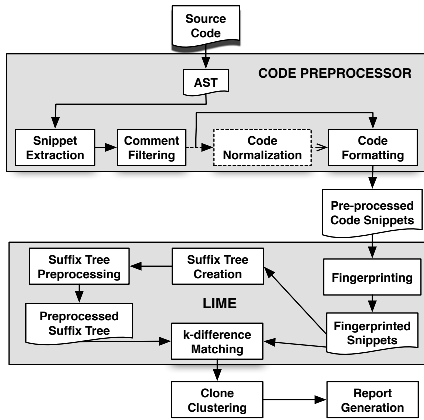

Fig. 2: Clone detection procedure [21]

Our tool also augments Eclipse's search engine by introducing the facility (Figure 3) to find all exact and near-miss cloned copies within a chosen boundary (selected files, packages/directories, projects, or the entire workspace) for any code fragment selected in the editor.

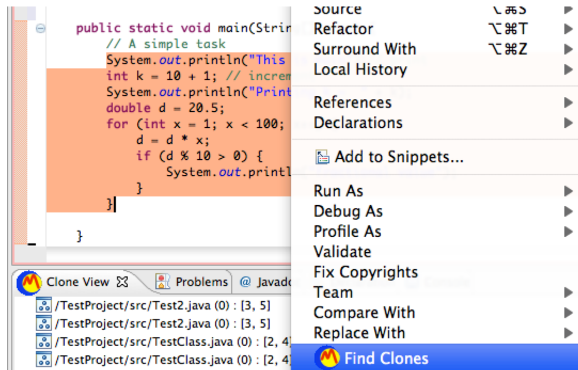

Fig. 3: User interface for clone search

B. Scheduling of Code Clone Refactoring

Refactoring of code clone is a significant part of effective clone management. To prevent code inflation and reduce maintenance cost, the amount of code clones should often be minimized by applying active refactoring. When the code clones in a given software system needs to be refactored, so many clones may appear to be potential candidates for refactoring. Moreover, the effort required for refactoring certain code clone, as well as the consequences (e.g., expected benefits, risks) are dependent on the type of clones and the underlying context. In addition, there may be sequential dependencies and conflicts among the refactoring activities. These lead to the necessity that, from all refactoring candidates a subset of non-conflicting refactoring activities be selected and ordered (for application) such that the quality of the codebase is maximized while the required effort is minimized [21]. However, such a problem is known to be NP-hard [3], [11], [12].

In this regard, we introduced an *effort model* for the estimation of code clone refactoring efforts. We also introduced a *constraint programming (CP)* approach for efficient scheduling code clone refactoring activities.

1) Code Clone Refactoring Effort Model: The effort model captures the efforts for context understanding, source code modification and navigation across the codebase. The *context understanding effort* is further realized by the efforts for understanding method delegation and inheritance hierarchy. The *source code modification effort* is estimated in terms of the efforts for token modification and code relocation. Finally, the *navigation effort* captures the dispersion of class hierarchy as well as the file-system hierarchy of the underlying codebase. More detail about the effort model can be found elsewhere [22].

2) Scheduling of Code Clone Refactoring: We identified three types of hard constraints (i.e., mutual exclusion, mutual inclusion, and sequential dependency) pertaining to scheduling of code clone refactoring. A priority scheme was also introduced to capture as soft constraints the practitioners' preferences on refactoring certain code clones. We formulated the problem of scheduling refactoring activities as a constraint satisfaction optimization problem and introduced a CP model for computation of the optimal solutions. Note that, an optimal solution is expected to make optimal balance along three dimensions: minimization of efforts while maximizing quality gain and satisfaction of priorities. Further detail can be found elsewhere [22].

3) Case Study for Evaluation: The CP model of the refactoring scheduler was implemented using OPL (Optimization Programming Language) in the IBM ILOG CPLEX Optimization Studio 12.2 IDE. Then we conducted a case study [22] for evaluation. For estimation of the effect of code quality, we used the QMOOD [2] metric suite. The study involved scheduling the refactoring of code clones in four software systems written in Java. We compared the performance of our CP approach with manual scheduling as well as variants of greedy approaches. Our CP scheduler was found to have outperformed all those approaches, as can be observed from the results presented in Table I and Figure 4.

TABLE I: Comparison of CP and greedy scheduling [22]

Subject systems	Scheduling approaches	Values at dimensions Prior.	Effort	Quality	Refac. chosen
Mutation Frame-work	Greedy^p	20.06	21.94	18.53	40
	Greedy^e	9.63	6.06	10.04	20
	Greedy^q	18.16	21.82	19.64	42
	CP	9.34	7.86	11.48	20
LIME	Greedy^p	22.42	21.12	19.93	47
	Greedy^e	13.00	8.28	13.61	33
	Greedy^q	16.29	23.49	26.07	51
	CP	11.04	12.32	16.12	33
gCad	Greedy^p	19.65	21.62	20.00	41
	Greedy^e	9.61	9.53	11.57	28
	Greedy^q	12.05	23.48	25.98	44
	CP	6.69	15.19	17.99	28
VisCad	Greedy^p	36.14	32.57	25.71	66
	Greedy^e	16.12	18.63	13.20	40
	Greedy^q	29.02	33.81	34.32	72
	CP	15.33	15.78	21.90	40

Here, Greedy^p = approach greedy towards priority satisfaction
Greedy^e = approach greedy towards effort minimization
Greedy^q = approach greedy towards quality gain

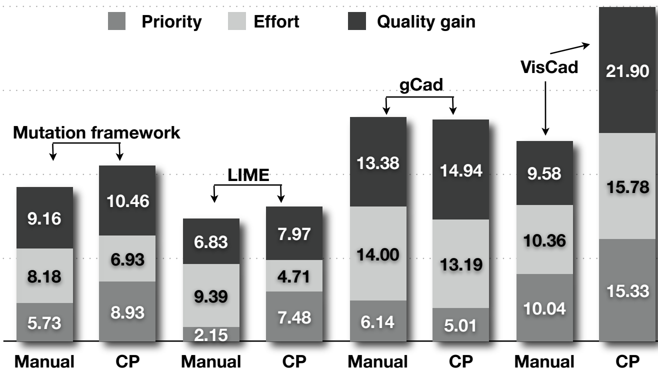

Fig. 4: Automated CP vs. manual scheduling [22]

IV. Contributions and Future Work

The high level analysis, presented in Section II-A, was the largest study on code clone evolution, at the time when the study was conducted [20]. Moreover, the study analyzed *Type-1* and *Type-2*, *Type-3* clones, while most of the previous work included *Type-1* and/or *Type-2* clones only. The low level analysis, described in Section II-B, was an exploratory extension to the work of Kim et al. [8], who first coined the notion of *clone genealogy*. Our study was on a larger number of diverse systems written in various programming languages, while theirs was on two small Java systems only. Most of the earlier work [5], [9], [10], [14] including that of Kim et al. [8] investigated clone evolution across subsequent CVS commit transactions or CVS snapshots over weekly intervals or so, whereas both of our studies were at the release level.

With regards to code clone refactoring, ours is the first *effort model* for object-oriented code, which, with minor tuning, can also be applied to procedural codebase [22]. Besides, we are the first to have applied the CP technique in scheduling code clone refactoring [22]. We are going to integrate the refactoring scheduler to the clone management system [21] we have been developing. Our immediate future work includes development of an integrated tool for automatic estimation of refactoring efforts based on our effort model. This will be followed by support for semi-automated *extraction of refactoring candidates* and *applications of refactoring operations*, along with seamless support for *edit propagation* and *simultaneous editing* of code clones. We strongly believe that our clone management tool, once completed, will be of immense help to the community in dealing with both exact and near-miss clones.

Acknowledgment: This work is supported in part by the Natural Science and Engineering Research Council of Canada (NSERC).

References

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

[2] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Engg.*, 28(1): 4–17, 2002.

[3] S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo. A Novel Approach to Optimize Clone Refactoring Activity. In *GECCO*, July 8 –12, 2006.

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

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

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

[7] 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.

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

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

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

[11] H. Liu, G. Li, Z. Ma, and W. Shao. Conflict-aware schedule of software refactorings. *IET Softw.*, 2(5): 446–460, 2008.

[12] S. Lee, G. Bae, H. S. Chae, and D. Bae, and Yong Rae Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Softw. Pract. Exper.*, Wiley Online Library, 2010.

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

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

[15] 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.

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

[17] 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.

[18] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Sc. of Com. Prog.*, 74: 470–495, 2009.

[19] 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.

[20] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy. Analyzing and Forecasting Near-miss Clones in Evolving Software: An Empirical Study. In *ICECCS*, pp. 295–304, 2011.

[21] M. F. Zibran and C. K. Roy. Towards Flexible Code Clone Detection, Management, and Refactoring in IDE. In *IWSC*, pp. 75–76, 2011.

[22] M. F. Zibran and C. K. Roy. A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring. In *SCAM*, 10 pp., 2011 (to appear).