An Empirical Study on Clone Stability

Manishankar Mondal Chanchal K. Roy Kevin A. Schneider
Department of Computer Science,
University of Saskatchewan, Canada
{mshankar.mondal, chanchal.roy, kevin.schneider}@usask.ca

ABSTRACT

Code cloning is a controversial software engineering practice due to contradictory claims regarding its effect on software maintenance. Code stability is a recently introduced measurement technique that has been used to determine the impact of code cloning by quantifying the changeability of a code region. Although most existing stability analysis studies agree that cloned code is more stable than non-cloned code, the studies have two major flaws: (i) each study only considered a single stability measurement (e.g., lines of code changed, frequency of change, age of change); and, (ii) only a small number of subject systems were analyzed and these were of limited variety.

In this paper, we present a comprehensive empirical study on code stability using four different stability measuring methods. We use a recently introduced hybrid clone detection tool, NiCAD, to detect the clones and analyze their stability in different dimensions: by clone type, by measuring method, by programming language, and by system size and age. Our in-depth investigation on 12 diverse subject systems written in three programming languages considering three types of clones reveals that: (i) cloned code is generally less stable than non-cloned code, and more specifically both Type-1 and Type-2 clones show higher instability than Type-3 clones; (ii) clones in both Java and C systems exhibit higher instability compared to the clones in C# systems; (iii) a system's development strategy might play a key role in defining its comparative code stability scenario; and, (iv) cloned and non-cloned regions of a subject system do not follow any consistent change pattern.[1]

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, Reverse Engineering and Reengineering.*

General Terms

Measurement and Experimentation

Keywords

Software Clones, Types of Clones, Code Stability, Modification Frequency, Changeability, Overall Instability.

[1]This work is based on an earlier work: SAC '12 Proceedings of the 2012 ACM Symposium on Applied Computing, Copyright 2012 ACM 978-1-4503-0857-1/12/03. http://doi.acm.org/10.1145/2245276.2231969.

1. INTRODUCTION

Frequent copy-paste activity by programmers during software development is common. Copying a code fragment from one location and pasting it to another location with or without modifications cause multiple copies of exact or closely similar code fragments to co-exist in software systems. These code fragments are known as clones [22, 25]. Whatever may be the reasons behind cloning, the impact of clones on software maintenance and evolution is of great concern.

The common belief is that the presence of duplicate code poses additional challenges to software maintenance by making inconsistent changes more difficult, introducing bugs and as a result increasing maintenance efforts. From this point of view, some researchers have identified clones as "bad smells" and their studies showed that clones have negative impact on software quality and maintenance [8, 15, 16, 19]. On the other hand, there has been a good number of empirical evidence in favour of clones concluding that clones are not harmful [1, 7, 10, 11, 27]. Instead, clones can be useful from different points of views [9].

A widely used term to assess the impact of clones on software maintenance is *stability* [7, 12, 13, 15]. In general, *stability of a particular code region measures the extent to which that code region remains stable (or unchanged) during the evolution of a software system.* If cloned code is more stable (changes less frequently) as compared to non-cloned code during software evolution, it can be concluded that cloned code does not significantly increase maintenance efforts. Different studies have defined and evaluated stability from different viewpoints which can be broadly divided into two categories:

(1) Stability measurement in terms of changes: Some methodologies [7, 12, 15, 6] have measured stability by quantifying the changes to a code region using two general approaches: (i) determination of the ratio of the number of lines added, modified and deleted to the total number of lines in a code region (cloned or non-cloned) [12, 15, 6] and (ii) determination of the frequency of modifications to the cloned and non-cloned code [7] with the hypothesis that the higher the modification frequency of a code region is the less stable it is.

(2) Stability measurement in terms of age: This approach [13] determines the average last changed dates of cloned and non-cloned code of a subject system. The hypothesis is that the older the average last change date of a code region is, the more stable it is.

To measure the comparative stability of cloned and non-cloned code, Krinke carried out two case studies [12, 13].

In his first study, he calculated the comparative instabilities caused by cloned and non-cloned regions in terms of addition, deletion and modification in these regions whereas in a most recent study [13] (elaborated in Section 3.2), he determined the average last change dates of the regions. Both of these studies suggest cloned code to be more stable than non-cloned code.

Hotta et al., in a recent study [7], calculated the modification frequencies of cloned and non-cloned code and found that the modification frequency of non-cloned code is higher than that of cloned code.

1.1 Motivation

Though all of the studies [7, 12, 13, 6] generally agreed on the higher stability of cloned code over non-cloned code, we conducted our research to address the following drawbacks of these studies.

(1) The studies only considered limited aspects of stability.

(2) General decisions were made without considering a wide variety of subject systems.

(3) No study addresses the comparative stabilities as well as impacts of different clone types. This issue is important in the sense that different types of clones might have different impacts (good or bad) on maintenance. Based on impact variability, we might take care of some specific clone types while leaving others alone.

(4) Instability of clones on the basis of language variability was not measured. This information might be very important from a managerial perspective. Projects developed using programming languages that exhibit high clone instability may require more care regarding cloning activities.

(5) Different studies were conducted on different experimental setups (e.g. different subject systems, different clone detection tools with different parameters, considering software releases or revisions at different intervals), which might be a potential cause behind their contradictory outcomes.

(6) No study performed statistical tests about how significant is the difference between the metric values of cloned and non-cloned code. If the metric values regarding cloned and non-cloned code are not significantly different, then we do not need to be much worried about clones.

(7) The existing metrics are not sufficient to reveal all aspects of changeability (as well as stability) of cloned and non-cloned code.

1.2 Contribution

Focusing on the issues mentioned above, our study contributes in the following ways.

(1) Considering the count of lines affected in source code modifications (additions, deletions or changes) we propose a new method which calculates four new metrics: UP (Unstable Proportion), UPHL (Unstable Proportion per 100 LOC), PCRM (Proportion of Code Region Modified), and OICR (Overall Instability of Code Region) that provide us more precise information about code changeability in comparison with existing metrics as well as methods [12, 6] that considered line counts. The comparison between our proposed metrics and the existing ones has been elaborated in Section 3.5. We investigated these metrics to compare the stability of cloned and non-cloned code.

(2) We have investigated four methods in total by implementing them on the same experimental setup and answered seven research questions as listed in Table 1. One of the four methods is our proposed new method that has already been mentioned in the previous point. Two of these methods are existing and were proposed by Hotta et al. [7] and Krinke [13]. The last method is our proposed variant of Krinke's method [13]. The reasons behind introducing this variant are elaborated in Section 3.3. The research questions (Table 1) belong to five different dimensions. The first three research questions are answered by investigating the metrics calculated by our proposed new method. The other four questions are answered by combining the experimental results of all four candidate methods.

For detecting clones, we used the recently introduced hybrid clone detection tool NiCad [3] that combines the strengths and overcomes the limitations of both text-based and AST-based clone detection techniques and exploits novel applications of a source transformation system to yield highly accurate identification of Type-1, Type-2 and Type-3 cloned code in software systems [20].

Our experimental results on three clone types of 12 subject systems written in three languages (Java, C and C#) reveal that:

(1) Cloned code gets modified significantly more often (supported with Mann-Whitney Wilcoxon (MWW) tests [17]) than non-cloned code.

(2) A significantly higher proportion of cloned LOC is modified in commit operations compared to non-cloned LOC.

(3) Type-1 and Type-2 clones are potential threats to a system's stability while Type-3 clones are possibly not.

(4) Clones in Java and C systems exhibit a higher level of instabilities as compared to those of C# systems. This is also statistically supported by Fisher's Exact Test [5].

(5) Cloned code generally exhibits higher instability than non-cloned code. However, cloned and non-cloned regions of subject systems do not follow any consistent change pattern. Moreover, the development strategy may have a strong impact on the stability of cloned and non-cloned code.

The rest of the paper is organized as follows. Section 2 outlines the relevant research. Section 3 elaborates on the candidate methods. Our experimental setup is described in Section 4 and Section 5 contains the experimental results. A detailed analysis of the experimental results is presented in Section 6 while Section 7 mentions some possible validity threats of our study. Section 8 concludes the paper and describes future directions. The work presented in this paper is an extended version of our earlier work [18].

2. RELATED WORK

Over the last several years, the impact of clones has been an area of focus for software engineering research resulting in a significant number of studies and empirical evidence. Kim et al. [10] introduced a clone genealogy model to study clone evolution and applied the model on two medium sized Java systems. They showed that: (i) refactoring of clones

Table 1: Research Questions

	Research Questions (RQs)	Dimensions
RQ1	How often is a particular code region type (cloned or non-cloned) modified during evolution? Which code region type is modified more often?	Comparative stability centric
RQ2	Which code region type (cloned or non-cloned) has a higher proportion of LOC modifications in the commit operations?	
RQ3	Which code region type (cloned or non-cloned) exhibits higher instability compared to the other?	
RQ4	Do different types of clones exhibit different stability scenarios? Which type(s) of clones is (are) more vulnerable to the system's stability?	Type centric
RQ5	Why and to what extent do the decisions made by different methods on the same subject system differ?	Method centric
RQ6	Do different programming languages exhibit different stability scenarios?	Language centric
RQ7	Is there any effect of system sizes and system ages on the stability of cloned and non-cloned code?	System centric

may not always improve software quality and (ii) immediate refactoring of short-lived clones is not required and that such clones might not be harmful. Saha et al. [27] extended their work by extracting and evaluating code clone genealogies at the release level of 17 open source systems and reported that most of the clones do not require any refactoring effort in the maintenance phase.

Kapser and Godfrey [9] strongly argued against the conventional belief of harmfulness of clones by investigating different cloning patterns. They showed that: (i) about 71% of the cloned code has a kind of positive impact in software maintenance and (ii) cloning can be an effective way of reusing stable and mature features in software evolution. Lozano and Wermelinger et al. performed three studies [14, 15, 16] on the impact of clones on software maintenance considering method level granularities using CCFinder [2]. They developed a prototype tool *CloneTracker* [14] to investigate the changeability of clones. The other studies, though conducted on a small number of Java systems (4 in [15] and 5 in [16]), reported that clones have a harmful impact on the maintenance phase because clones often increase maintenance efforts and are vulnerable to a system's stability.

Juergens et al. [8] studied the impact of clones on large scale commercial systems and suggested that: (i) inconsistent changes occurs frequently with cloned code and (ii) nearly every second unintentional inconsistent change to a clone leads to a fault. Aversano et al. [1] on the other hand, carried out an empirical study that combines the clone detection and co-change analysis to investigate how clones are maintained during evolution or bug fixing. Their case study on two subject systems confirmed that most of the clones are consistently maintained. Thummalapenta et al. [28] in another empirical study on four subject systems reported that most of the clones are changed consistently and other inconsistently changed fragments evolve independently.

In a recent study [6] Göde et al. replicated and extended Krinke's study [12] using an incremental clone detection technique to validate the outcome of Krinke's study. He supported Krinke by assessing cloned code to be more stable than non-cloned code in general.

Code stability related research conducted by Krinke [12, 13] and Hotta et al. [7] referred to in the introduction is elaborated on in the next section.

In our empirical study, we have replicated Krinke's [13] and Hotta et al.'s [7] methods and implemented our variant of Krinke's method [13] and our proposed new method using NiCad [3]. Our experimental results and analysis of those results reveal inportant information about comparative stabilities and harmfulness of three clone types along with language based stability trends.

3. STABILITY MEASURING METHODS

This section discusses the three methods and associated metrics that we have implemented for our investigation. These methods follow different approaches and calculate different metrics but their aim is identical in the sense that each of these methods takes the decision about whether cloned code of a subject system is more stable than its non-cloned code. For this reason we perform a head-to-head comparison of the stability decisions indicated by the metrics of these three methods and focus on the implementation and strategic differences that cause decision dissimilarities.

3.1 Modification Frequencies

Hotta et al. [7] calculated two metrics: (i) MF_d (Modification Frequencies of Duplicate code) and (ii) MF_n (Modification Frequencies of Non-Duplicate code) considering all the revisions of a given codebase extracted from SVN. Their metric calculation strategy involves identification and checking out of relevant revisions of a subject system, normalization of source files by removing blank lines, comments and indents, detection and storing of each line of duplicate code into the database. The differences between consecutive revisions are also identified and stored in the database. Then, MC_d (Modification Count in Duplicate code region) and MC_n (Modification Count in Non-Duplicate code region) are determined exploiting the information saved in the database and finally MF_d and MF_n are calculated using the following equations [7]:

$$MF_d = \frac{\sum_{r \epsilon R} MC_d(r)}{|R|} * \frac{\sum_{r \epsilon R} LOC(r)}{\sum_{r \epsilon R} LOC_d(r)} \quad (1)$$

$$MF_n = \frac{\sum_{r \epsilon R} MC_n(r)}{|R|} * \frac{\sum_{r \epsilon R} LOC(r)}{\sum_{r \epsilon R} LOC_n(r)} \quad (2)$$

Here, R is the number of revisions of the candidate subject system. $MC_d(r)$ and $MC_n(r)$ are the number of modifications in the duplicate and non-duplicate code regions respec-

tively between revisions r and $(r+1)$. MF_d and MF_n are the modification frequencies of the duplicate and non-duplicate code regions of the system. $LOC(r)$ is the number of LOC in revision r. $LOC_d(r)$ and $LOC_n(r)$ are respectively the numbers of duplicate and non-duplicate LOCs in revision r.

3.2 Average Last Change Date

Krinke [13] introduced a new concept of code stability measurement by calculating the average last change dates of cloned and non-cloned regions of a codebase using the *blame* command of SVN. He considers only a single revision (generally the last revision) unlike the previous method proposed by Hotta et al. [7] that considers all the revisions up to the last one. The *blame* command on a file retrieves each line's revision and date when the line was last changed. He calculates the average last change dates of cloned and non-cloned code from the file level and system level granularities.

File level metrics: (1) Percentage of files where the average last change date of cloned code is older than that of non-cloned code (PF_c) (cloned code is older than non-cloned code) in the last revision of a subject system. **(2)** Percentage of files where the average last change date of cloned code is newer than that of non-cloned code (PF_n) (cloned code is younger than non-cloned code) in the last revision of a subject system.

System level metrics: (1) Average last change date of cloned code (ALC_c) for the last revision of a candidate subject system. **(2)** Average last change date of non-cloned code (ALC_n) for the last revision of a candidate subject system. To calculate file level metrics in our implementation, we considered only the analyzable source files, which excludes two categories of files from consideration: (i) files containing no cloned code and (ii) fully cloned files. But, system level metrics are calculated considering all source files. According to this method, the older the code is the more stable it is.

Calculation of average last change date: Suppose five lines in a file correspond to 5 revision dates (or last change dates) 01-Jan-11, 05-Jan-11, 08-Jan-11, 12-Jan-11, 20-Jan-11. The average of these dates was calculated by determining the average distance (in days) of all other dates from the oldest date 01-Jan-11. This average distance is $(4+7+11+19)/4 = 10.25$ and thus the average date is 10.25 days later to 01-Jan-11 yielding 11-Jan-11.

3.3 Proposed Variant of Krinke's Method

We have proposed a variant of Krinke's methodology [13] to analyze the longevity (stability) of cloned and non-cloned code by calculating their average ages. We also have used the *blame* command of SVN to calculate the age for each of the cloned and non-cloned lines in a subject system.

Suppose we have several subject systems. For a specific subject system we work on its last revision R. By applying a clone detector on revision R, we can separate the lines of each source file into two disjoint sets: (i) containing all cloned lines and (ii) containing all non-cloned lines. Different lines of a file contained in R can belong to different previous revisions. If the *blame* command on a file assigns the revision r to a line x, then we understand that line x was produced in revision r and has not been changed up to the last revision R. We denote the revision of x as $r = revision(x)$. The creation date of r is denoted as $date(r)$. In the last revision R, we can determine the age (in days) of this line by the following equation:

$$age(x) = date(R) - date(revision(x)) \quad (3)$$

We have calculated the following two average ages for cloned and non-cloned code from system level granularity.

1. Average age of cloned code (AA_c) in the last revision of a subject system. This is calculated by considering all cloned lines of all source files of the system.

2. Average age of non-cloned code (AA_n) in the last revision of a subject system. AA_n is calculated by considering all non-cloned lines of all source files of the system.
According to our method, a higher average age is the implication of higher stability.

We have introduced this variant to address the following issues in Krinke's method.

1. *blame* command of SVN gives the revisions as well as revision dates of all lines of a source file including its comments and blank lines. Krinke's method does not exclude blank lines and comments from consideration. This might play a significant role on skewing the real stability scenario.

2. As indicated in the average last change date calculation process, Krinke's method often introduces some rounding errors in its results. This might force the average last change dates of cloned and non-cloned code to be equal (There are examples in Section 5).

3. The method's dependability on the file level metrics sometimes alters the real stability scenario. The type-3 case of 'Greenshot' is an example where both Hotta et al.'s method and our proposed variant make a similar decision (non-cloned code is more stable); but, the file level metrics of Krinke's method alters this decision. The system level metrics (ALCs) of Krinke's method could not make a stability determination because the metrics corresponding to cloned (ALC_c) and non- cloned (ALC_n) code were the same.

Our proposed variant overcomes these issues while calculating stability results. It does not calculate any file level metrics because its system level metrics are adequate in decision making. It should also be mentioned that Hotta et al.'s method also ensures the exclusion of blank lines and comments from consideration through some preprocessing steps prior to clone detection.

3.4 Proposed New Method and Metrics

Hotta et al.'s method [7] calculates modification frequency by only considering the count of modifications that occurred in a subject system without considering the number of lines affected by those modifications. Focusing on this issue, we propose a new method that calculates four new metrics for measuring the stability (as well as changeability) of a particular code region. The descriptions and calculation procedures of the metrics are given below.

UP (Unstable Proportion): Unstable proportion (UP) of a particular code region (cloned or non-cloned) is the proportion of the commit operations in which that code region is modified.

Suppose C is the set of all commit operations through which a subject system has evolved. The two sets of commit op-

erations in which the cloned and non-cloned regions of this system was modified are C_c and C_n respectively. We should note that the sets C_c and C_n might not be disjoint. The unstable proportions of cloned and non-cloned regions are calculated using the following two equations, respectively.

$$UP_c = \frac{100 \times |C_c|}{|C|} \quad (4)$$

$$UP_n = \frac{100 \times |C_n|}{|C|} \quad (5)$$

Here, UP_c is the unstable proportion of cloned code and UP_n is the unstable proportion of non-cloned code.

UPHL (Unstable Proportion per 100 LOC): Generally the UP of a particular code region (cloned or non-cloned) is positively proportional to the total LOC of that region. UP_n is expected to be higher than the corresponding UP_c for a particular subject system. To determine how often a particular code region is modified, we should also consider the total LOC of that region. From this perspective we calculated the *UPHL* using equations Eq. 6 and Eq. 7. According to the equations, the *UPHL* of a particular region is equal to the UP per 100 LOC of that region. In other words, *UPHL determines the likelihood of being modified per 100 LOC of a code region.* As there are many revisions of a particular software system, we determined the total LOC of the cloned or non-cloned region of this system by calculating the average LOC per revision of the corresponding region.

$$UPHL_c = \frac{100 \times UP_c \times |C|}{\sum_{c_i \epsilon C} LOC_c c_i} \quad (6)$$

$$UPHL_n = \frac{100 \times UP_n \times |C|}{\sum_{c_i \epsilon C} LOC_n c_i} \quad (7)$$

Here, $UPHL_c$ and $UPHL_n$ are the unstable proportions per 100 LOC of the cloned and non-cloned regions respectively. $LOC_c(c_i)$ is the count of total cloned LOC of the revision corresponding to the commit c_i. Also, $LOC_n(c_i)$ is the count of total non-cloned LOC of the revision corresponding to the commit c_i.

PCRM (Proportion of Code Region Modified): For a particular code region (cloned or non-cloned) we calculate the PCRM by determining the proportion of that code region getting modified in commit operations. Here, we consider only those commit operations where the particular code region had some modifications. Considering the previous example, we can calculate the *PCRM* of cloned and non-cloned code according to the following equations.

$$PCRM_c = \frac{100 \times \sum_{c_i \epsilon C_c} LC_c(c_i)}{\sum_{c_i \epsilon C_c} LOC_c(c_i)} \quad (8)$$

$$PCRM_n = \frac{100 \times \sum_{c_i \epsilon C_n} LC_n(c_i)}{\sum_{c_i \epsilon C_n} LOC_n(c_i)} \quad (9)$$

Here, $PCRM_c$ and $PCRM_n$ are respectively the proportions of cloned and non-cloned regions that are modified. $LC_c(c_i)$ and $LC_n(c_i)$ are the number of lines changed in cloned and non-cloned regions in commit operation c_i.

OICR (Overall instability of code region): We calculate the OICR of a particular code region (cloned or non-cloned) by multiplying its UP with its PCRM. We see that the PCRM determines the proportion of a particular code region being modified per commit operation and UP determines the proportion of commit operations in which that particular code region is being modified. Thus, the multiplication of these two will determine the instability exhibited by the code region throughout the evolution. We calculate OICR for cloned and non-cloned code according to the following two equations.

$$OICR_c = UP_c \times PCRM_c \quad (10)$$

$$OICR_n = UP_n \times PCRM_n \quad (11)$$

Here, $OICR_c$ and $OICR_n$ are respectively the overall instabilities of cloned and non-cloned regions of a software system.

3.5 Comparison of Our Proposed Metrics With Existing Ones

Two existing studies performed by Krinke [12] and Göde and Harder [6] investigated some metrics that are similar to our proposed metric PCRM. Krinke [12] computed the instabilities of cloned and non-cloned code with respect to addition, deletion, and change. Göde and Harder extended Krinke's study [12] considering tokens instead of lines. While Göde and Harder analyzed all of the revisions of a particular software system, Krinke considered 200 weekly snapshots for each of his candidate systems. However, none of these studies could answer the first and second research questions (RQ 1 and RQ 2 in Table 1).

RQ 1 was not addressed because no existing study defined and evaluated a relevant metric. Our proposed metric *UPHL* addresses this question.

Also, RQ 2 was not addressed by any existing studies. Our metric *PCRM* answers this question. The studies performed by Krinke [12] and Göde and Harder [6] computed some related metrics. However, the strategic difference between our proposed metric (PCRM) and the existing ones is that while calculating the metric value for a particular code region we considered only those commits where that particular code region was modified excluding those commits where that region did not have any modifications. However, the existing studies did not exclude commits that do not have any effect on a particular code region while calculating the metric value for the region.

We think that if a particular commit does not affect a particular code region, we should not consider that commit for investigating the stability of that region, because that commit is not responsible for increasing the instability of that region. We call such a commit an *ineffective commit* for that particular region. As the previous studies [12, 6] did not exclude these ineffective commits for a particular region while investigating the region's instability, no study could determine what proportion of code from a particular region is being modified when that region is actually receiving some changes (due to effective commits). Our proposed metric PCRM eliminates this drawback of the similar existing metrics.

3.6 Major Difference Between Hotta's Method and Our Proposed New Method

Hotta et al.'s method relies on the count of modifications for making stability decisions disregarding the number of lines affected by the modifications. However, our proposed new method relies on the count of affected lines. In a particular commit operation, l ($l > 0$) consecutive lines of a particular code region might get modified. According to Hotta et al.'s method the count of modifications is one. It calculates modification frequencies by considering this modification count disregarding the number of lines modified. However, our proposed new method calculates PCRM and OICR considering the count of affected lines (for this example l).

This difference may cause disagreements in the stability decisions made by the two methods. Suppose the cloned and non-cloned regions of a software system received respectively m_c and m_{nc} modifications in a commit operation. The total number of lines affected by these m_c and m_{nc} modifications are l_c and l_{nc} respectively. If $m_c > m_{nc}$, Hotta et al.'s method will decide that cloned code is more unstable. However, in this situation our proposed new method may decide the opposite, since l_{nc} could be greater than l_c.

3.7 Major Difference Between Hotta's Method and the Method Proposed by Krinke and Its Variant

Hotta et al.'s method [7] considers *all* modifications to a region from its creation and it does not matter *when* the modifications to the region are applied. The other two methods only consider the *last* modification (which can also be creation) and do not consider any modification before.

Suppose a file contains two lines denoted by x and y in revision 1 and this file passed through 100 commits during which x had 5 changes and y had only one change. Let the change on y occur at the 99th commit and the last change on x occur at the 50th commit. A *blame* command on the last revision (100) of this file will assign x revision 50 and y will be assigned revision 99. According to both Krinke's method and our variant, x is older than y because the revision date corresponding to revision 50 is much older than the revision date corresponding to revision 99 and thus, x will be suggested to be more stable than y by these two methods. On the other hand, the method proposed by Hotta et al. counts the number of modifications that occurred on these two lines. Consequently, Hotta et al. will suggest y to be more stable than x because the modification frequency of x will obviously be greater than that of y.

4. EXPERIMENTAL SETUP

We implemented all four candidate methods in a common framework in Java using MySQL for the back-end database. Instead of using any existing implementations, we have reimplemented the two already existing methods (proposed by Hotta et al.[7] and Krinke [13]) as we wanted to have a common framework for comparison. Our selected subject systems and setup of the clone detection tool are described below.

4.1 Clone Detection

We used the NiCad [3, 21] clone detection tool to detect clones in the subject systems in our study. NiCad can detect both exact and near-miss clones at the function or block level of granularity. We detected block clones with a minimum size of 5 LOC in the pretty-printed format that removes comments and formatting differences.

NiCad can provide clone detection results in two ways: (1) by separating three types of clones (Type-1, Type-2, Type-3) and (2) by combining all three types of clones. The NiCad settings for detecting the three types of clones is provided in Table 2. The dissimilarity threshold means that the clone fragments in a particular clone class may have dissimilarities up to that particular threshold value between the pretty-printed and/or normalized code fragments. We set the dissimilarity threshold to 20% with blind renaming of identifiers for detecting Type-3 clones. For all these settings above NiCad was shown to have high precision and recall [20]. We have used NiCad's combined type results for answering the first three research questions. The remaining four questions have been answered by investigating the three types of clones separately.

Table 2: NiCad Settings

Clone Types	Identifier Renaming	Dissimilarity Threshold
Type 1	none	0%
Type 2	blindrename	0%
Type 3	blindrename	20%

Table 3: Subject Systems

	Systems	Domains	LOC	Rev
Java	DNSJava	DNS protocol	23,373	1635
	Ant-Contrib	Web Server	12,621	176
	Carol	Game	25,092	1699
	Plandora	Project Management	79,853	32
C	Ctags	Code Def. Generator	33,270	774
	Camellia	Image Processing	100,891	608
	QMail Admin	Mail Management	4,054	317
	Hashkill	Password Cracker	83,269	110
C#	GreenShot	Multimedia	37,628	999
	ImgSeqScan	Multimedia	12,393	73
	Capital Resource	Database Management	75,434	122
	MonoOSC	Formats and Protocols	18,991	355

Rev = Revisions

4.2 Subject Systems

Table 3 lists the details of the subject systems used in our study. We selected these subject systems because they are diverse in nature, differ in size, span 11 application domains, and are written in three programming languages. Also, most of these systems differ from those included in the studies of Krinke[13] and Hotta et al.[7], which was intentionally done to retrieve exact stability scenarios.

5. EXPERIMENTAL RESULTS

For answering the first three research questions we applied our proposed new method on the combined type clone detec-

Table 4: Decision Making Strategy

Method	Metrics		Decision Making	
	CC	NC	CC More Stable	NC More Stable
Proposed Method	$OICR_c$	$OICR_n$	$OICR_c$ $<OICR_n$	$OICR_n$ $<OICR_c$
Hotta et al. [7]	MF_d	MF_n	MF_d $<MF_n$	MF_n $<MF_d$
Krinke [13]	ALC_c, PF_c	ALC_n, PF_n	ALC_c is older	ALC_n is older
Krinke's Variant	AA_c	AA_n	AA_c $>AA_n$	AA_n $>AA_c$
Special Decision for Krinke's Method				
if $ALC_c = ALC_n$ then				
$PF_c > PF_n$ implies CC more stable				
$PF_n > PF_c$ implies NC more stable				
CC = Cloned Code		NC = Non-cloned Code		

tion results of each of the 12 subject systems and obtained the values of the four new metrics. However, for answering the remaining questions we applied each of the four methods on each of the three types of clones of each of the 12 candidate systems and calculated the metric values for three types of clones separately. By applying our proposed new method on the individual type results we obtained the value of the fourth metric (OICR) only. So, in our investigation regarding the research questions 4 to 7, each of the four methods contributed one metric (4 metrics in total).

We should mention that we have three different implementations corresponding to three types of clones for each of the candidate methods. Thus, we have 12 (4 subject systems × 3 clone types) stability evaluation systems in total. For answering the RQs 4 to 7, we applied each of these 12 stability evaluation systems on each of the 12 subject systems to get the values of the stability metrics. So, we have 144 (12 subject systems × 12 stability evaluation systems) sets of metric values from 144 different experiments. Each set contains two values: (1) the metric value for cloned code (of a particular type), and (ii) the metric value for non-cloned code (corresponding to that particular type). From these two values, we can make a decision about comparative stability of cloned and non-cloned code. For this reason, we have called each of these metric value sets a decision point. Finally, our investigation regarding the RQs 4 to 7 depends on these 144 decision points obtained by conducting 144 different experiments. However, for answering the RQs 1 to 3 we conducted 12 experiments (by applying our proposed new method on the combined type results of 12 subject systems). The following paragraphs in this section describes the tables that contain the results obtained from the experiments.

Table 5 shows the average last change dates obtained by applying Krinke's method. Table 7 and Table 9 contain respectively the modification frequencies and average ages of cloned and non-cloned code. File level metrics for two special cases (Table 4) are shown in Table 6. The overall instabilities of cloned and non-cloned code obtained by applying our proposed new method are presented in Table 8. Interpretation of the table data is explained below.

Almost all of the tables are self-explanatory. Decision making strategies for Tables 5, 7, 9, and 8 are elaborated in Table 4. However, for our proposed new method we have specified only one metric *Overall Instability of Code Region* (out of four) in Table 4. The other three metrics are investigated in section 6.1.

Table 6: File Level Metrics for Two Systems

Subject System	Clone Type	PF_c	PF_n
Plandora	Type-2	6	4
Greenshot	Type-3	43	12

Table 7: Modification Frequencies of Cloned (MF_d) and Non-cloned (MF_n) code by Hotta et al.'s method

	Systems	Type 1		Type 2		Type 3	
		MF_d	MF_n	MF_d	MF_n	MF_d	MF_n
Java	DNSJava	21.61	7.12	19.34	6.99	7.93	8.66
	Ant-Contrib	3.62	1.49	2.02	1.52	1.43	1.59
	Carol	8.15	6.60	4.07	3.69	9.91	8.97
	Plandora	0.44	0.92	0.45	0.97	0.55	1.11
C	Ctags	6.37	3.82	7.19	7.17	6.71	3.68
	Camellia	18.50	18.04	42.37	17.73	30.02	17.53
	QMailAdmin	5.09	2.74	8.83	5.47	8.24	2.58
	Hash Kill	61.24	115.22	59.92	115.64	65.75	118.04
C#	GreenShot	7.94	6.07	6.92	6.07	8.13	6.06
	ImgSeqScan	0	20.93	0	21.06	0	21.29
	Capital Resource	0	67.15	0	67.31	3.63	67.11
	MonoOSC	8.58	29.14	7.92	29.23	10.62	29.63

The stability decisions (as per Table 4) of all the metric values contained in the Tables 5, 7, 8, and 9 are summarized in Table 10, which contains decisions for 144 (12 subject systems x 4 methods x 3 clone types) decision points corresponding to 144 cells containing stability decision symbols ('⊕' and '⊖', explained in the table).

For decision making regarding Krinke's method we prioritized the system level metrics (ALC_c and ALC_n) as they represent the exact scenarios of the whole system. There are only two examples of special cases as per Table 4: (i) Type-3 case of 'Greenshot' and (ii) Type-2 case of 'Plandora'. For these, the system level metrics (Table 5) are the same and thus, we based the decisions on the file level metrics. We show the file level metrics for these two cases in Table 6 without providing them for all 36 cases (12 subject systems x 3 clone types).

6. ANALYSIS OF RESULTS

We present our analysis of the experimental results in five dimensions and answer the seven research questions introduced in Table 1.

To address the first three research questions we produced four graphs: Fig. 1, Fig. 2, Fig. 3, and Fig. 4. Table 11 contains 36 (12 subject systems, 3 clone types) decision points and was developed from Table 10 for the purpose of answering research questions 4 to 7. Each cell in the table corresponds to a decision point and implies agreement ('⊕' or '⊖') or disagreement ('⊗') of the candidate methods regarding stability decisions. The meanings of '⊕', '⊖' and '⊗' are provided in the tables.

Table 5: Average Last Change Dates of Cloned (ALC_c) and Non-cloned (ALC_n) code

	Clone Types	Type 1		Type 2		Type 3	
	Systems	ALC_c	ALC_n	ALC_c	ALC_n	ALC_c	ALC_n
Java	DNSJava	24-Mar-05	26-Apr-04	21-Jan-05	24-Apr-04	31-Mar-05	19-Apr-04
	Ant-Contrib	22-Sep-06	03-Aug-06	18-Sep-06	02-Aug-06	08-Sep-06	03-Aug-06
	Carol	25-Nov-07	18-Jan-07	25-Nov-07	14-Jan-07	12-Jun-05	27-Feb-07
	Plandora	31-Jan-11	01-Feb-11	01-Feb-11	01-Feb-11	31-Jan-11	01-Feb-11
C	Ctags	27-May-07	31-Dec-06	24-Mar-07	31-Dec-06	17-Sep-06	01-Jan-07
	Camellia	04-Nov-07	14-Nov-07	17-Jul-08	14-Nov-07	8-Feb-09	9-Nov-07
	QMail Admin	07-Nov-03	24-Oct-03	19-Nov-03	24-Oct-03	26-Nov-03	24-Oct-03
	Hash Kill	14-Jul-10	02-Dec-10	27-Jul-10	02-Dec-10	19-Jul-10	02-Dec-10
C#	GreenShot	11-Jun-10	21-Jun-10	12-Jun-10	21-Jun-10	20-Jun-10	20-Jun-10
	ImgSeqScan	19-Jan-11	14-Jan-11	17-Jan-11	14-Jan-11	19-Jan-11	14-Jan-11
	Capital Resource	13-Dec-08	12-Dec-08	11-Dec-08	12-Dec-08	10-Dec-08	12-Dec-08
	MonoOSC	08-Apr-09	21-Mar-09	05-Mar-09	21-Mar-09	21-Jan-09	22-Mar-09

Table 8: Overall Instabilities of Cloned ($OICR_c$) and Non-cloned ($OICR_n$) code by our proposed new method

		Type 1		Type 2		Type 3	
	Systems	$OICR_c$	$OICR_n$	$OICR_c$	$OICR_n$	$OICR_c$	$OICR_n$
Java	DNSJava	12.41	12.62	20.85	16.67	21.21	16.56
	Ant-Contrib	37.16	7.28	3.65	8.53	10.61	7.94
	Carol	6.06	8.64	5.31	8.79	12.41	8.00
	Plandora	10.5	3.72	4.99	3.83	6.17	3.36
C	Ctags	4.63	9.37	9.72	9.33	5.90	9.65
	Camellia	16.34	5.31	24.84	5.29	25.60	8.56
	QMailAdmin	49.84	32.37	44.77	32.38	52.50	30.72
	Hash Kill	7.42	53.69	0.71	53.44	11.75	55.23
C#	GreenShot	10.76	10.18	8.27	10.29	10.63	10.24
	ImgSeqScan	0	243.35	0	246.23	0	247.34
	CapitalResource	0	9.88	0	9.78	2.05	9.83
	MonoOSC	7.28	37.32	8.82	36.96	12.56	37.89

Table 9: Average Age in days of Cloned (AA_c) and Non-cloned (AA_n) code by the proposed variant

		Type 1		Type 2		Type 3	
	Systems	AA_c	AA_n	AA_c	AA_n	AA_c	AA_n
Java	DNSJava	2181	2441	2247	2443	2210.9	2446.9
	Ant-Contrib	853.6	903.7	896.1	903.3	870.6	904.4
	Carol	189.6	210.9	190.3	211.3	227	209.6
	Plandora	51.82	51.32	50.6	51.4	51.5	51.32
C	Ctags	1301.4	1345.2	1351.9	1345	1564.8	1343.4
	Camellia	1066.8	1056.7	810.9	1057.3	604.9	1062.4
	QMail Admin	2664.2	2678.1	2651.7	2678.2	2644.6	2678.3
	Hash Kill	261.5	118.5	250.3	118.4	257.9	118
C#	Green Shot	103.1	97.1	102.9	97.1	94.5	97.2
	ImgSeq Scan	14	20	15.6	20.3	14.4	20.4
	Capital Resource	86.7	86.5	88	86.5	89.3	86.5
	Mono OSC	315.4	313.5	347.9	313	378	312.3

Table 10: Comparative Stability Scenarios

	Methods / Systems	Krinke [13]			Hotta et al.[7]			Variant			Proposed new		
		T1	*T2*	*T3*	*T1*	*T2*	*T3*	*T1*	*T2*	*T3*	*T1*	*T2*	*T3*
Java	DNSJava	⊖	⊖	⊖	⊖	⊖	⊕	⊖	⊖	⊖	⊕	⊖	⊖
	Ant-Contrib	⊖	⊖	⊖	⊖	⊖	⊕	⊖	⊖	⊖	⊖	⊕	⊖
	Carol	⊖	⊖	⊕	⊖	⊖	⊖	⊖	⊖	⊕	⊕	⊕	⊖
	Plandora	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊖	⊕	⊖	⊖	⊖
C	Ctags	⊖	⊖	⊕	⊖	⊖	⊖	⊖	⊕	⊕	⊕	⊖	⊕
	Camellia	⊕	⊖	⊖	⊖	⊖	⊖	⊕	⊖	⊖	⊖	⊖	⊖
	QMailAdmin	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖
	Hash Kill	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
C#	GreenShot	⊕	⊕	⊕	⊖	⊖	⊖	⊕	⊕	⊖	⊖	⊕	⊖
	ImgSeqScan	⊖	⊖	⊖	⊕	⊕	⊕	⊖	⊖	⊖	⊕	⊕	⊕
	CapitalResource	⊖	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
	MonoOSC	⊖	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕

⊕=*Cloned Code More Stable*
⊖=*Non-Cloned Code More Stable*
T1, T2 and T3 denote clone types 1, 2 and 3 respectively

Table 11: Overall stability decisions by methods

Lang.	**Java**				**C**				**C#**			
Clone Types	DNSJava	Ant-Contrib	Carol	Plandora	Ctags	Camellia	QMail Admin	Hash Kill	GreenShot	ImgSeqScan	Capital Resource	MonoOSC
Type-1	⊖	⊖	⊖	⊕	⊖	⊗	⊖	⊕	⊗	⊗	⊕	⊕
Type-2	⊖	⊖	⊖	⊗	⊖	⊖	⊖	⊕	⊕	⊗	⊕	⊕
Type-3	⊖	⊖	⊗	⊕	⊕	⊖	⊖	⊕	⊖	⊗	⊕	⊕

⊕=*Most of the methods agree with* ⊕
⊖=*Most of the methods agree with* ⊖
⊗=*Decision conflict (Two methods agree with* ⊕ *and the remaining two methods agree with* ⊖*)*

For example, in Table 10 three methods (excluding our proposed new method) agree there is higher Type-1 clone instability in 'Ctags'. For the Type-3 case in 'Carol', two methods (the method proposed by Krinke and the proposed variant of Krinke's method) agree there is higher cloned code stability, whereas the other two methods agree there is higher non-cloned code stability. Thus, in Table 11, Type-1 clones for 'Ctags' is marked with '⊖' and Type-3 clones for 'Carol' is marked with ⊗.

6.1 Analysis Regarding the Comparative Stability of Cloned and Non-cloned Code

6.1.1 Analysis regarding UP and UPHL

From the graph in Fig.1 presenting the unstable proportions of cloned and non-cloned code we see that the unstable proportion of non-cloned code is always higher than that of cloned code. This is obvious because a larger code region would generally require more modifications to be maintained. To get more precise information we determined the UPHL for cloned and non-cloned regions of each of the candidate systems. The comparative scenario between $UPHL_c$ and $UPHL_n$ is presented in Fig. 2.

According to this graph, for most of the subject systems (11 out of 12) $UPHL_c > UPHL_n$. Thus, *cloned code is more likely to be modified compared to non-cloned code.* In answer to the first research question (RQ 1) we can say that *cloned code is modified more often than non-cloned code.*

We performed the MWW (Mann-Whitney Wilcoxon) test [17] on the observed values ($UPHL_c$ and $UPHL_n$) for the 11 subject systems with a higher $UPHL_c$ to determine whether $UPHL_c$ is significantly higher than $UPHL_n$. The two tailed probability value (p-value) for this test is 0.00244672. The p-value is much less than 0.05 and it implies that $UPHL_c$ is significantly higher than $UPHL_n$ for our candidate subject systems. Thus, according to our experimental result, *cloned code is modified significantly more often than non-cloned code.*

6.1.2 Analysis regarding PCRM

The graph in Fig. 3 presents the comparison between the $PCRM_c$ and $PCRM_n$ of the each of the candidate systems. We see that for most of the subject systems (10 out of 12), $PCRM_c > PCRM_n$. For one (ImgSeqScan) of the remaining two subject systems, $PCRM_c = 0$ because the cloned regions of this subject system did not have any modifications. Thus, in answer to the second research question (RQ 2) we can say that *the proportion of cloned regions (i.e., the proportion of cloned LOC) modified due to effective commits is generally higher than the proportion of the non-cloned regions getting modified due to effective commits.*

Considering the 10 subject systems with higher $PCRM_c$ we performed the MWW (Mann-Whitney Wilcoxon) test [17] on the observed values of $PCRM_c$ and $PCRM_n$ to determine whether $PCRM_c$ is significantly higher than $PCRM_n$ for these systems. The two tailed probability value (p-value) regarding the test is 0.01854338. We see that the p-value is less than 0.05. Thus, the difference between $PCRM_c$ and $PCRM_n$ is marginally significant according to our findings.

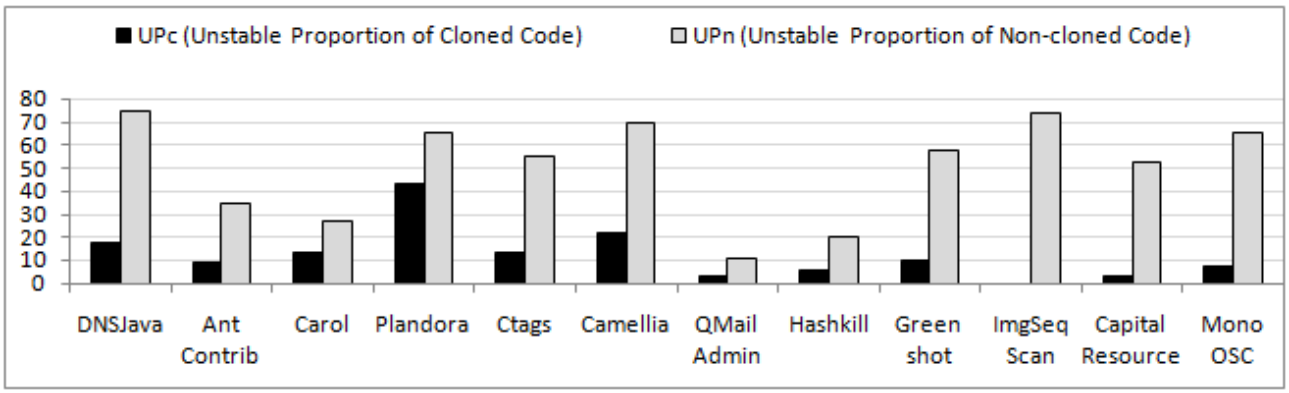

Figure 1: Unstable proportions (UP) of cloned and non-cloned code

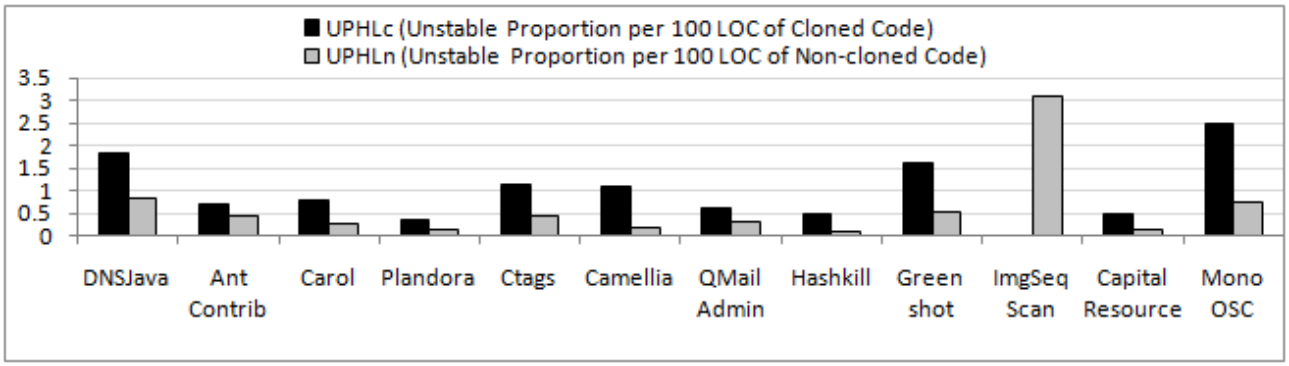

Figure 2: Unstable proportion per 100 LOC (UPHL) of cloned and non-cloned code

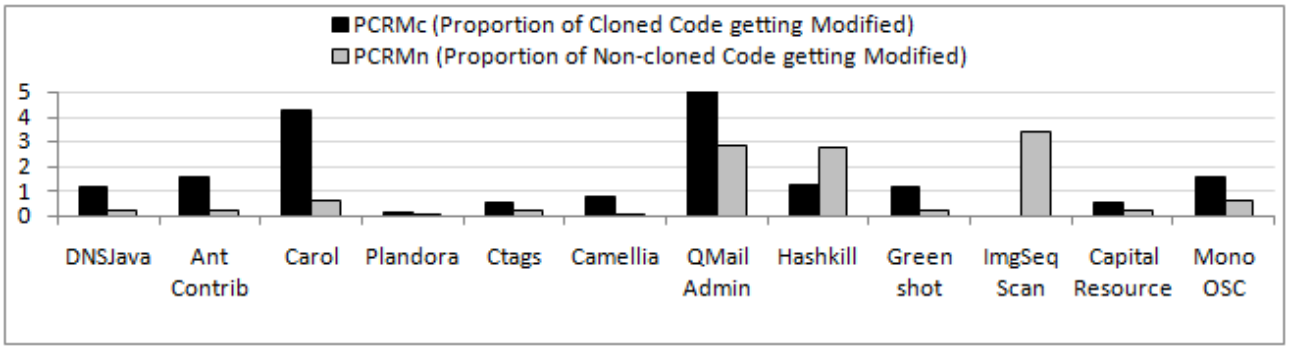

Figure 3: Proportions of cloned and non-cloned regions getting modified (PCRM)

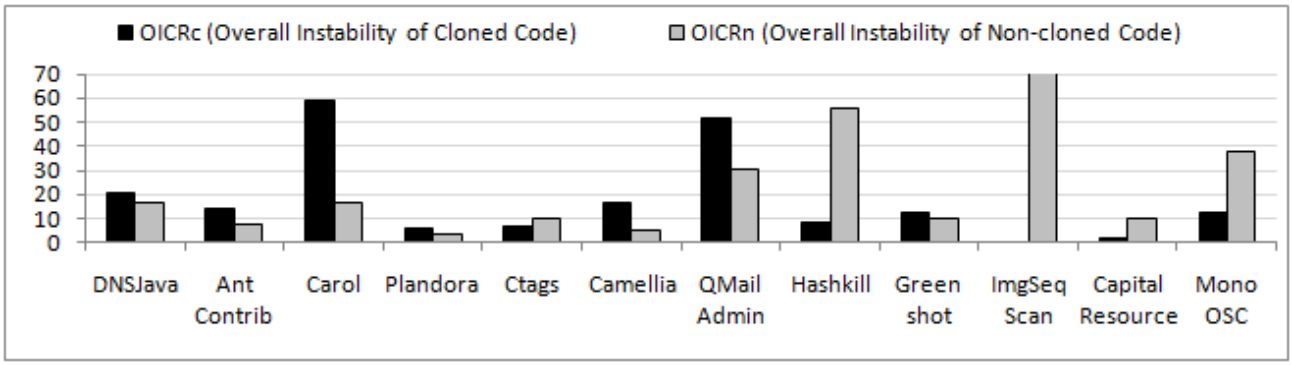

Figure 4: Overall instabilities of cloned and non-cloned code (OICR)

6.1.3 Analysis regarding OICR

The comparison of the overall instabilities of the cloned and non-cloned regions of the candidate systems is presented in Fig. 4. According to this graph seven subject systems have higher overall instability of cloned code (higher $OICR_c$) while the remaining five subject systems have the opposite. For one (ImgSeqScan) of these remaining five systems, $OICR_c$ equals zero because cloned regions of this system did not get modified. In other words, the cloned regions of this system did not have any effective commits. However, the comparison scenario presented by the graph indicates that *in the presence of effective commits, overall instability of a cloned region is generally higher than that of a non-cloned region.*

6.1.4 Analysis Result

From the analysis of our experimental results regarding the comparative stability of cloned and non-cloned code, we see that (1) cloned code gets modified significantly more often than non-cloned code, (2) the proportion of cloned region getting modified due to effective commits (defined in the last paragraph of Section 3.5) is higher than that of non-cloned region, and (3) finally, in answer to the third research question (RQ 3) we can say that the overall instability of cloned code is generally higher than that of non-cloned code. The comparative scenario implies that software clones are generally less stable than non-cloned code and thus, clones should be managed with proper care to increase the stability of software systems.

6.2 Type Centric Analysis

In this analysis, we tried to answer the fourth research question (Table 1) by investigating how a particular method's decisions on a particular subject system vary with the variation of clone types. We have the following observations.

The stability decisions made by a method on a specific subject system corresponding to three types of clones are similar for 31 cases with some minor variations for the remaining cases. That is, Table 10, shows there are 64.58% similar cases among 48 cases. Each case consists of three decisions for three types of clones made by a particular method on a particular subject system. As an example of variations, consider the decisions made by Hotta's method for 'DNSJava'. For the Type-3 case (Table 7), $MF_d < MF_n$ suggests that Type-3 clones are more stable than the corresponding non-cloned code. However, according to this method, Type-1 and Type-2 clones of 'DNSJava' are much less stable than non-cloned code (the difference of MFs for the Type-3 case is smaller compared to the differences for the other two cases). We analyzed the experimental results in the following two ways.

6.2.1 Analysis 1

This analysis is based on the agreement-disagreement scenario of the Table 11. According to the agreed decisions points (Table 11) of the Type-1 case:

(i) clones decrease the stability of a system with probability = No. of cells with cloned code less stable/total no. of cells = 5/12 = 0.42.

(ii) non-cloned code decreases the stability of a system with probability = 4/12 = 0.33.

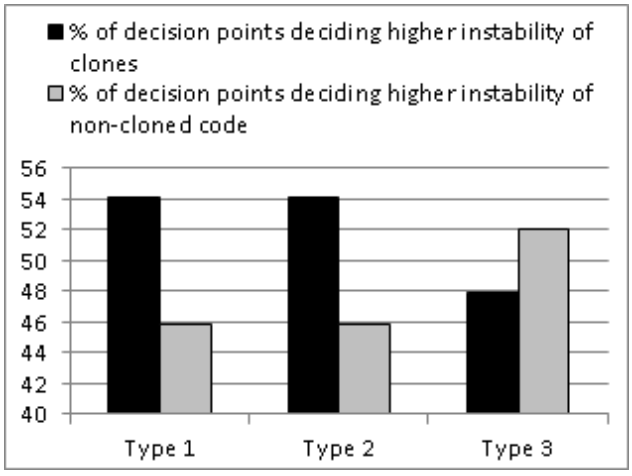

Figure 5: Type centric analysis

For the Type-2 case, these two probabilities are 0.50 (for cloned code) and 0.33 (for non-cloned code) respectively. So, for both of these cases (Type-1 and Type-2) cloned code has a higher probability of decreasing the system's stability. But, for Type-3 case these two probabilities are the same (0.42 for both cloned and non-cloned code). We see that for both Type-1 and Type-2 cases, clones have higher probability of making a system unstable compared to the probability of corresponding non-cloned code. However, Type-3 clones do not appear to be more unstable than non-cloned code according to our findings.

6.2.2 Analysis 2

In this case, we analyzed the data in Table 10. In this table, each type of clones contribute 48 decision points in total. Considering these 48 decision points (for each type of clones) we calculated the proportions of decision points agreeing with higher instability of cloned or non-cloned code. These proportions are plotted in Fig. 5.

According to this graph, the higher proportion of decision points belonging to both Type-1 and Type-2 case agree with the higher instability of cloned code compared to the Type-3 case. Thus, Type-1 and Type-2 clones are more vulnerable in the software systems compared to the vulnerability exhibited by Type-3 clones.

6.2.3 Analysis Result

Both Type-1 clones (created by exact copy-paste activities), and Type-2 clones (created by renaming identifiers and changing data types) should be given more attention during the development and maintenance phase.

6.3 Method Centric Analysis

We see that Table 10 contains 144 decision points where each method contributes 36 decisions (corresponding to 12 systems and 3 clone types). From this we can retrieve the decision making scenario presented in Table 12 exhibited by the candidate methods. According to this table, the majority of the methods (three out of four) agree there is higher instability in cloned code. However, there are decision disagreements among the methods. The disagreements have been analyzed in the following way.

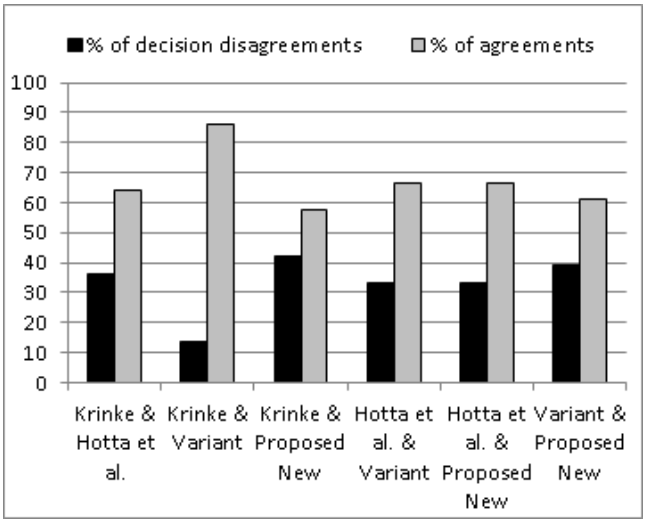

Figure 6: Method centric analysis regarding disagreements

6.3.1 Analysis of disagreements

We see that in Table 10 each method contributes 36 decision points. From this table we determined the percentage of disagreements for each pair of methods. For each decision point belonging to a particular method we get a corresponding decision point in another method. For each pair of methods, we see whether the corresponding decision points are conflicting (making different stability decisions) or not. From 36 sets of corresponding points for a particular method pair, we determine the proportion of conflicting sets. These proportions are shown in the graph of Fig.6. We have the following observations from the graph.

(1) We see that the method proposed by Krinke and its variant have the lowest proportion of conflicts. As both of these methods work on the last revision of a subject system they should exhibit a higher proportion of agreements in their decisions. The reason behind the observed disagreements is that the proposed variant excludes blank lines and comments from consideration while calculating average ages of cloned and non-cloned code. But, Krinke's method is biased by the blank lines and comments because it does not exclude these from consideration while determining average last change dates.

(2) Each of the methods proposed by Krinke and its variant has strong disagreements with each of the other two methods (proposed by Hotta et al. and our proposed new method). The reason behind this disagreement is that while both of the methods proposed by Hotta et al. and our proposed new method examine each of the existing revisions of a particular software system, the other two methods examine only the last revision for making stability decisions. In the following example we explain an observed strong disagreement.

Example and explanation of a strong disagreement: We consider 'ImqSeqScan' as an (extreme) example. For each clone type, each of the methods proposed by Hotta et al. and the proposed new method shows strong disagreement to the decision of Krinke's method and its variant. Each of the three types of clones was suggested to be more stable than non-cloned code by the methods proposed by Hotta et al. (Table 7) and the proposed new method (Table 8). However, both Krinke's method and its variant yield the opposite decisions (Table 5 and 9). More interestingly, both Hotta et al.'s method and our proposed new method reveal that the cloned regions of 'ImgSeqScan' did not receive any change (modification frequencies of cloned code is 0 according to the Table 7, overall instability of cloned code is 0 according to the Table 8) during the entire lifetime (consisting of 73 commit transactions) where the other two methods show that the cloned code is significantly younger. In this case the regions of cloned code have only been created lately and have not been modified after creation. The following explanation will clarify this.

Table 12: Stability w.r.t. candidate methods

Decision Parameters	**% of Decision Points**			
	Krinke [13]	Hotta et al.[7]	Proposed variant	Proposed New
Non-cloned code more stable (cloned code less stable)	55.56	52.78	52.78	47.22
Cloned code more stable	44.44	47.22	47.22	52.78

Suppose a subject system receives 100 commit transactions. Some clone fragments were created in some of these commits but no existing clone fragment was modified at all. In such a case, both Hotta et al.'s method and our proposed new method will see that there are no modifications in the cloned region. As a result, MF_d (for Hotta et al.'s method) and $OICR_c$ (for the proposed new method) will be zero. On the other hand, the *blame* command will retrieve the creation dates of the clone fragments existing in the last revision of the system and Krinke's method will determine the average last change date for the cloned region considering these creation dates. If the creation dates of some clone fragments are newer than the modification dates of non-cloned fragments which forces the average last change date of the cloned region to be newer than that of the non-cloned region, Krinke's method will suggest cloned code to be less stable than non-cloned code. Thus, the cloned or non-cloned region of a subject system might be represented to be less stable than its counterpart even if it does not undergo any modifications during the entire evolution time while its counterpart does.

(3) The proposed new method disagrees with Hotta et al.'s method for 33.33% cases. The main reason behind this disagreement has already been explained in Section 3.6. Plandora is an extreme example of such disagreement. According to Hotta et al.'s method, each type of clones of this subject system are more stable than the corresponding non-cloned code. But, our proposed new method makes the opposite decision in each case.

Finally, in answer to the fifth research question (RQ 5) we can say that the stability decisions made by the candidate methods are often not similar and both Hotta et al.'s method and our proposed new method have strong disagreements with the other two methods in many cases.

Table 13: Stability w.r.t. programming languages

Decision Parameters	% of Agreed Decision Points		
	Java	C	C#
Non cloned code more stable (Cloned code less stable)	66.67	58.33	8.33
Cloned code more stable	16.67	33.33	58.33
Conflicting decisions	16.66	8.34	33.34

Table 14: Fisher's Exact Tests for prog. languages

	Java	C	Java	C#	C	C#
CCLS	67	58	67	8	58	8
NCLS	17	33	17	58	33	58
DC	17	8	17	33	8	33
P = 0.0103			P <0.0001		P <0.0001	

CCLS = Cloned Code Less Stable
NCLS = Non-cloned Code Less Stable
DD = Decision Conflicts

6.3.2 Analysis result

Considering all candidate methods and metrics we see that cloned code (all three types) has a higher probability to force a system into an unstable state compared to non-cloned code. According to Table 10, cloned code is less stable than non-cloned code for 75 cells (among 144 cells). The opposite is true for the remaining 69 cells. So the probability by which cloned code makes the system unstable is 75/144 = 0.52 which outweighs the probability of non-cloned code (0.48). Though the difference between the probabilities is very small, it disagrees with the conclusion drawn by both Krinke [13] and Hotta et al.[7] regarding comparative stability. Thus, clones should be carefully maintained and refactored (if possible) instead of keeping aside.

6.4 Language Centric Analysis

We performed language centric analysis in two ways.

6.4.1 Analysis 1

This analysis is based on the agreement-disagreement scenario of Table 11. Our set of subject systems consists of four systems from each of the three languages (Java, C and C#). In Table 11, each language contributes 12 (4 subject systems, 3 clone types) decision points. Considering these decision points we retrieved the language specific stability scenario presented in Table 13.

According to this table, both Java and C exhibit higher cloned code instability: 66.67% and 58.33% of the cases, respectively. The majority of the candidate methods agreed there is higher instability in cloned code. An exactly the opposite scenario was observed for C#. Moreover, for C# we can observe the highest proportion (33.33%) of decision conflicts. Thus, clones in both Java and C systems have a higher probability of making a system unstable compared to the clones in C# systems. Our Fisher's Exact Test results regarding the language centric statistics are described below.

Fisher's Exact Test: We performed Fisher's exact tests [5] on the three possible paired-combinations of the three languages using the values in Table 13 to see whether there are significant differences among the observed proportions of different languages. We defined the following null hypothesis. The values in Table 13 were rounded before using in Fisher's exact test.

Null Hypothesis: There is no significant difference between the stability scenarios presented by different programming languages.

From Table 14 we see the *P value* for each paired combination of programming languages is less than 0.05. This rejects the null hypothesis and confirms that there are significant differences among the observed scenarios of different programming languages.

6.4.2 Analysis 2

This analysis is based on the Table 10. In this table we see that each method contributes 12 decision points for each programming language. For each combination of method and language we determined two proportions: (1) the proportion of decision points agreeing there is higher cloned code instability, and (2) the proportion of decision points agreeing there is higher non-cloned code instability. These proportions are presented in Fig. 7.

In the bar chart (Fig. 7) we see that for each method, a higher proportion of decision points belonging to both Java and C agree there is higher cloned code instability. The opposite scenario is exhibited by C#. For each method, a higher proportion of decision points (belonging to C#) agree that there is higher cloned code stability. Thus, from this analysis we can also say that clones in both Java and C systems have a higher probability of making a system unstable compared to the clones in C#.

6.4.3 Analysis result

In answer to the sixth research question (RQ 6) we can say that clones in both Java and C systems exhibit significantly higher instability compared to the clones in C# systems and so developers as well as project managers should be more careful regarding clones during software development using these languages (Java and C).

6.5 System Centric Analysis

In the system centric analysis we investigated whether system sizes and system ages affect the comparative stabilities. In this investigation we wanted to observe how modifications occur in the cloned and non-cloned code of a subject system as the system becomes older and bigger in size. So, we recorded and plotted the modification frequencies of four subject systems for different revisions. We chose 'DNSJava', 'Carol', 'MonoOSC' and 'Hashkill' in this investigation. 'DNSJava' and 'Carol' have a large number of revisions compared to the revision numbers of other two systems. On the other hand 'Hashkill' is much bigger than the remaining three systems in terms of LOC. So, selecting these system we have a range of systems in terms of LOCs and revision numbers covering three languages. Also, these subject systems yielded contradictory stability scenarios for the method proposed by Hotta et al.

We present four line graphs (Fig. 8, Fig. 9, Fig. 10, Fig. 11) for these subject systems plotting their modification fre-

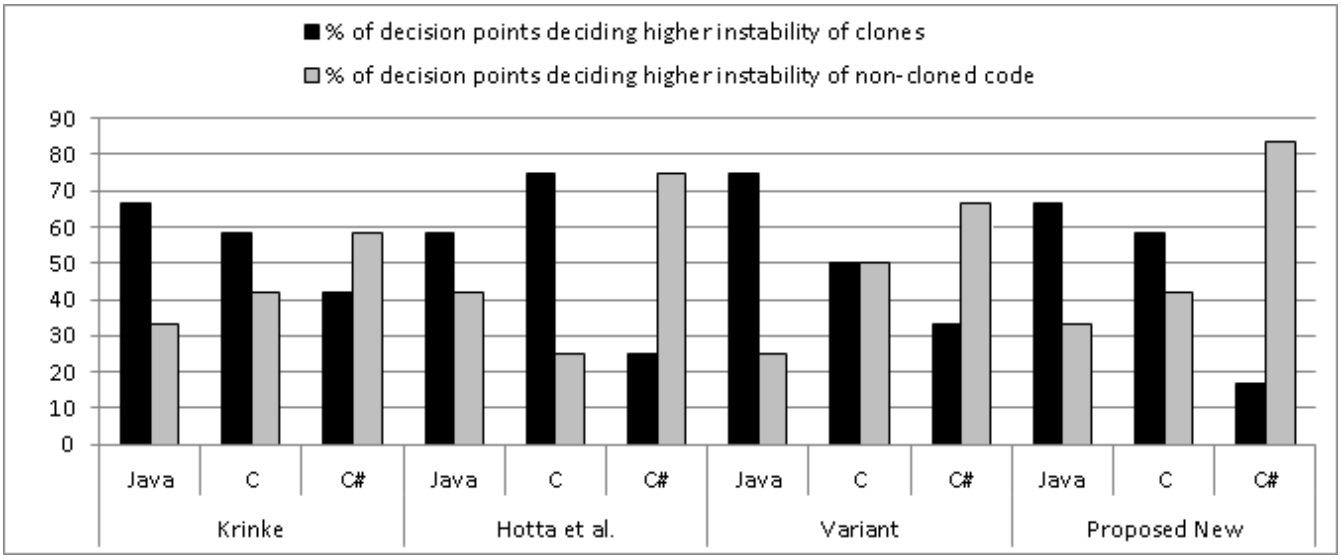

Figure 7: Language centric analysis

quencies for each of the revisions beginning with the second revision. For each revision r ($r \geq 2$), the modification frequencies of cloned and non-cloned code plotted in the graph were calculated by considering all revisions from 1 to r. The intention is to calculate the modification frequencies for r ($r \geq 2$) considering r as the current last revision. These visual representations of the gradual changes of modification frequencies reflect the exact trends of how the cloned and non-cloned regions were modified in the development phase.

In both Fig. 8 and Fig. 10, we see that for some development time cloned code was more stable than non-cloned code and vice versa. But the graph in Fig. 9 shows that for most of the development time, cloned code was modified more frequently than the non-cloned code. The graph in Fig. 11 exhibits a completely different scenario. The four graphs exhibit no change consistency or bias. Also, in the case of Carol (Fig. 9) we see that although for most of the life time the modification frequency curves showed opposite characteristics, the curves tend to meet each other at the end. On the other hand, the curves of the other three systems seem to diverge from each other. From Table 3 we can see that these four systems are of diverse sizes and nature. Thus, the convergence or divergence of the modification frequency curves is not dependent on the system sizes. So, RQ 7 can be answered by the observation that system sizes and system ages do not affect the stability of cloned and non-cloned code in a consistent or correlated way.

It is worth noting that every system can have a different development strategy which can affect changes to cloned and non-cloned code. For example, programmers might be afraid of changing cloned code because of the risk of inconsistent changes and would try to restrict the changes to the non-cloned code. Another possibility is that developers are advised to not change any code of other authors and thus are forced to create a clone in order to apply a change. However, such development strategies cannot be identified by looking at the change history alone and thus it is not possible to measure the impact on cloned and non-cloned code.

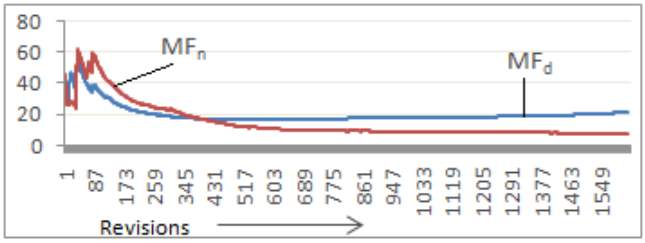

Figure 8: MFs for DNSJava (Type-1 case. Non-cloned code is more stable)

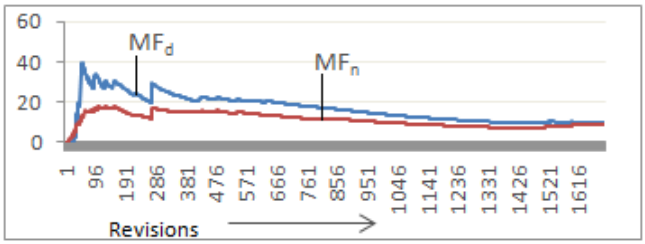

Figure 9: MFs for Carol (Type-3 case. Non-cloned code is more stable)

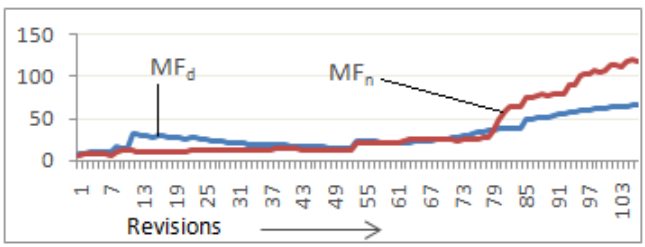

Figure 10: MFs for Hash Kill (Type-3 case. Cloned code is more stable)

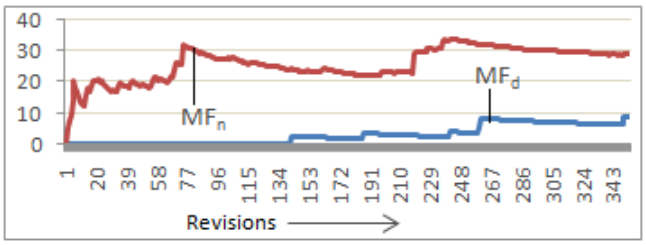

Figure 11: MFs for MonoOSC (Type-1 case. Cloned code is more stable)

7. THREATS TO VALIDITY

In the experimental setup section we mentioned the clone granularity level (block clones), difference thresholds and identifier renaming options that we have used for detecting three clone types. Different setups in corresponding clone types might result in different stability scenarios. However, the NiCad setups that we have used for detecting three types clones are considered standard [23, 24, 26, 4, 29] and thus we contend that the clone detection results that we have investigated are reliable.

8. CONCLUSION

In this paper we presented an in-depth investigation on the comparative stabilities of cloned and non-cloned code. We presented a five dimensional analysis of our experimental results to answer seven research questions. The ultimate aim of our investigation is to find out the changeabilities exhibited by different types of clones and languages and whether there is any yet-undiscovered consistency in code modification biasing the stability scenarios. We introduced a new method that calculates four new stability related metrics for the purpose of analysis.

According to our comparative stability centric analysis, cloned code is more unstable (as well as vulnerable) than non-cloned code because clones get modified significantly more often than non-cloned code (supported with Mann-Whitney-Wilcoxon tests). Also, the proportion of the cloned regions modified in effective commits is significantly higher than the proportion of non-cloned regions being modified.

However, our system centric analysis suggests that there are no existing biases in the modifications as well as stabilities of cloned and non-cloned code, and system development strategy can play an important role in driving comparative stability scenarios.

Our type centric analysis reveals that Type-1 (exact clones) and Type-2 (clones with differences in identifier names and data types) clones are possibly harmful for a system's stability. They exhibit higher probabilities of instabilities than the corresponding non-cloned code. Thus, these clone types should be given more attention both from a development and a management perspective.

Our language centric analysis discovers that clones of Java and C systems show higher modification probabilities compared to those of C# systems. This argument is also supported by statistical proof using Fisher's exact test (2 tailed).

Our method centric analysis discovers the causes of strong and weak disagreements of the candidate methodologies in making stability decisions. In this analysis we evaluated 144 decision points of comparative stabilities and found that cloned code exhibits higher changeability than that of non-cloned code which contradicts the already established bias ([13, 7]) regarding comparative stabilities of cloned vs. non-cloned code. Thus, cloned code is not necessarily stable as was observed in the previous studies [7, 13] and clones should be managed.

Our future plan is to perform an exhaustive empirical study for further analysis of the impacts of clones using several clone detection tools, methods and a wider range of subject systems.

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

9. REFERENCES

[1] Aversano, L., Cerulo, L., and Penta, M. D., "How clones are maintained: An empirical study", in Proc. *The 11th European Conference on Software Maintenance and Reengineering (CSMR)*, 2007, pp. 81-90.

[2] CCFinderX. `http://www.ccfinder.net/ccfinderxos.html`

[3] Cordy, J. R., and Roy, C. K., "The NiCad Clone Detector", in Proc. *The Tool Demo Track of the 19th International Conference on Program Comprehension (ICPC)*, 2011, pp. 219-220.

[4] Cordy, J. R., and Roy, C. K., "Tuning Research Tools for Scalability and Performance: The NICAD Experience", in *Science of Computer Programming*, 2012, 26 pp. (to appear)

[5] Fisher's Exact Test. `http://in-silico.net/statistics/fisher_exact_test/2x3`.

[6] Göde, N., and Harder, J., "Clone Stability", in Proc. *The 15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011, pp. 65-74.

[7] Hotta, K., Sano, Y., Higo, Y., and Kusumoto, S., "Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software", in Proc. *The Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)* , 2010, pp. 73-82

[8] Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S., "Do Code Clones Matter?", in Proc. *The 31st International Conference on Software Engineering (ICSE)*, 2009, pp. 485-495.

[9] Kapser, C., and Godfrey, M. W., ""Cloning considered harmful" considered harmful: patterns of cloning in software", in *Journal of Empirical Software Engineering*. 13(6), 2008, pp. 645-692.

[10] Kim, M, Sazawal, V., Notkin, D., and Murphy, G. C., "An empirical study of code clone genealogies", in Proc. *The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, 2005, pp. 187-196.

[11] Krinke, J., "A study of consistent and inconsistent changes to code clones", in Proc. *The 14th Working*

Conference on Reverse Engineering (WCRE), 2007, pp. 170-178.

[12] Krinke, J., "Is cloned code more stable than non-cloned code?", in Proc. *The 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2008, pp. 57-66.

[13] Krinke, J., "Is Cloned Code older than Non-Cloned Code?", in Proc. *The 5th International Workshop on Software Clones (IWSC)*, 2011, pp.28-33.

[14] Lozano, A., Wermelinger, M., and Nuseibeh, B., "Evaluating the Harmfulness of Cloning: A Change Based Experiment", in Proc. *The 4th International Workshop on Mining Software Repositories (MSR)*, 2007, pp. 18-21.

[15] Lozano, A., and Wermelinger, M., "Tracking clones' imprint", in Proc. *The 4th International Workshop on Software Clones (IWSC)*, 2010, pp. 65-72.

[16] Lozano, A., and Wermelinger, M., "Assessing the effect of clones on changeability", in Proc. *The 24th IEEE International Conference on Software Maintenance (ICSM)*, 2008, pp. 227-236.

[17] Mann-Whitney-Wilcoxon Test: http://elegans.som.vcu.edu/ leon/stats/utest.html

[18] Mondal, M., Roy, C. K., Rahman, M. S., Saha, R. K., Krinke, J., and Schneider, K. A., "Comparative Stability of Cloned and Non-cloned Code: An Empirical Study", in Proc. *The 27th Annual ACM Symposium on Applied Computing (SAC)*, 2012, pp. 1227–1234.

[19] Mondal, M., Roy, C. K., and Schneider, K. A., "Dispersion of Changes in Cloned and Non-cloned Code", in Proc. *The 6th International Workshop on Software Clones (IWSC)*, 2012, pp. 29-35 .

[20] Roy, C. K., and Cordy, J. R., "A mutation / injection-based automatic framework for evaluating code clone detection tools", in Proc. *The IEEE International Conference on Software Testing, Verification, and Validation Workshops* , 2009, pp. 157-166.

[21] Roy, C. K., and Cordy, J. R., "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization" in Proc. *The 16th IEEE International Conference on Program Comprehension (ICPC)*, 2008, pp. 172-181.

[22] Roy, C. K., Cordy, J. R., and Koschke, R., "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", in *Science of Computer Programming*, 74 (2009) 470-495, 2009.

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

[24] Roy, C. K., and Cordy, J. R., "An Empirical Evaluation of Function Clones in Open Source Software", in Proc. *The 15th Working Conference on Reverse Engineering (WCRE)*, 2008, pp. 81-90.

[25] Roy, C. K., and Cordy, J. R., "Scenario-based Comparison of Clone Detection Techniques", in Proc. *The 16th IEEE International Conference on Program Comprehension (ICPC)*, 2008, pp.153-162.

[26] Saha, R. K., Roy, C. K., and Schneider, K. A., "An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies", in Proc. *The 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 293-302.

[27] Saha, R. K., Asaduzzaman, M., Zibran, M. F., Roy, C. K., and Schneider, K. A., "Evaluating code clone genealogies at release level: An empirical study", in Proc. *The 10th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 2010, pp. 87-96.

[28] Thummalapenta, S., Cerulo, L., Aversano, L., and Penta, M. D., "An empirical study on the maintenance of source code clones", in *Journal of Empirical Software Engineering (ESE)*, 15(1), 2009, pp. 1-34.

[29] Zibran, M. F., Saha, R. K., Asaduzzaman, M., and Roy, C. K., "Analyzing and Forecasting Near-miss Clones in Evolving Software: An Empirical Study", in Proc. *The 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2011, pp. 295-304.

ABOUT THE AUTHORS:

Manishankar Mondal is a graduate student in the Department of Computer Science of the University of Saskatchewan, Canada under the supervision of Dr. Chanchal Roy and Dr. Kevin Schneider. He is a lecturer at Khulna University, Bangladesh and currently on leave for pursuing his higher studies. He received the Best Paper Award from the 27th Symposium On Applied Computing (ACM SAC 2012) in the Software Engineering Track. His research interests are software maintenance and evolution including clone detection and analysis, program analysis, empirical software engineering and mining software engineering.

Chanchal Roy is an assistant professor of Software Engineering/Computer Science at the University of Saskatchewan, Canada. While he has been working on a broad range of topics in Computer Science, his chief research interest is Software Engineering. In particular, he is interested in software maintenance and evolution, including clone detection and analysis, program analysis, reverse engineering, empirical software engineering and mining software repositories. He served or has been serving in the organizing and/or program committee of major software engineering conferences (e.g., ICSM, WCRE, ICPC, SCAM, ICSE-tool, CASCON, and IWSC). He has been a reviewer of major Computer Science journals including IEEE Transactions on Software Engineering, International Journal of Software Maintenance and Evolution, Science of Computer Programming, Journal of Information and Software Technology and so on. He received his Ph.D. at Queen's University, advised by James R. Cordy, in August 2009.

Kevin Schneider is a Professor of Computer Science, Special Advisor ICT Research and Director of the Software Engineering Lab at the University of Saskatchewan. Dr. Schneider has previously been Department Head (Computer Science), Vice-Dean (Science) and Acting Chief Information Officer and Associate Vice-President Information and Communications Technology. Before joining the University of Saskatchewan, Dr. Schneider was CEO and President of Legasys Corp., a software research and development company specializing in design recovery and automated software engineering. His research investigates models, notations and techniques that are designed to assist software project teams develop and evolve large, interactive and usable systems. He is particularly interested in approaches that encourage team creativity and collaboration.