Is Cloned Code Really Stable?

Manishankar Mondal   Md Saidur Rahman   Chanchal K. Roy   Kevin A. Schneider
University of Saskatchewan, Canada
{mshankar.mondal, saeed.cs, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—Clone has emerged as a controversial term in software engineering research and practice. The impact of clones is of great importance from software maintenance perspectives. Stability is a well investigated term in assessing the impacts of clones on software maintenance. If code clones appear to exhibit a higher instability (i.e., higher change-proneness) than non-cloned code, then we can expect that code clones require higher maintenance effort and cost than non-cloned code. A number of studies have been done on the comparative stability of cloned and non-cloned code. However, these studies could not come to a consensus. While some studies show that code clones are more stable than non-cloned code, the other studies provide empirical evidence of higher instability of code clones. The possible reasons behind these contradictory findings are that different studies investigated different aspects of stability using different clone detection tools on different subject systems using different experimental setups. Also, the subject systems were not of wide varieties.**

Emphasizing these issues (with several others mentioned in the motivation) we have conducted a comprehensive empirical study where we have - (i) implemented and investigated seven existing methodologies that explored different aspects of stability, (ii) used two clone detection tools (NiCad and CCFinderX) to implement each of these seven methodologies, and (iii) investigated the stability of three types (Type-1, Type-2, Type-3) of clones. Our investigation on 12 diverse subject systems covering three programming languages (Java, C, C#) with a list of 8 stability assessment metrics suggest that (i) cloned code is often more unstable (change-prone) than non-cloned code in the maintenance phase, (ii) both Type 1 and Type 3 clones appear to exhibit higher instability than Type 2 clones, (iii) clones in Java and C programming languages are more change-prone than the clones in C#, and (iv) changes to the clones in procedural programming languages seem to be more dispersed than the changes to the clones in object oriented languages. We also systematically replicated the original studies with their original settings and found mostly equivalent results as of the original studies. We believe that our findings are important for prioritizing code clones from management perspectives.

Keywords*——*Code Clones; Code Stability; Software Maintenance and Evolution.

I. Introduction

Reuse of code fragments with or without modifications by copying and pasting from one location to another is a common yet controversial software development practice. This results in the existence of exactly or nearly similar code fragments in different components of a software system. These code fragments are termed as clones. In addition to copy-paste activity, some other issues including programmers' behaviour like laziness and tendency to repeat common solutions, technology limitations, code evolvability, lack of code understandability and external business forces have influences on code cloning [29]. Whatever may be the causes behind cloning, the impacts of code clones are of great concern [8]–[10], [12], [21], [23], [24], [26], [27], [29]–[31], [34]–[38], [42], [46], [48], [49], [56] from the software maintenance point of view.

While a number of studies [8], [12], [21], [24], [26], [29]–[31], [46], [56] have identified some positive impacts of code clones, there is strong empirical evidence [9], [10], [23], [27], [34]–[37], [42], [47] of negative impacts of clones too. A widely investigated term to assess the impact of code clones on software maintenance is stability [19], [24], [32], [33], [36], [41]. According to the literature, stability of a particular code region measures the extent to which that code region is not likely to change (i.e., the extent to which that code region remains stable) during software evolution. So far, seven studies [19], [24], [33], [36], [37], [41], [42] have defined eight different stability measurement metrics. Each study has quantified and compared its respective stability metric for cloned and non-cloned code. The intuition is that if cloned code exhibits higher instability (i.e., change-proneness) than non-cloned code, then we can suggest cloning to be responsible for increased maintenance effort. However, these studies could not come to a consensus. Table I shows the eight stability measurement metrics with their respective implications.

From Table I we see that each of the eight existing metrics is directly related to differential changes in clone and non-clone regions. This supports the contention that the negative effects of clones in maintenance are directly related to the increased changes in source code. There are two main causes for the negative impacts of clones: (i) hidden bug propagation [34] and (ii) unintentional inconsistent changes [10], [21]. Let us first consider bug propagation. Suppose a code fragment contains a bug which is temporarily hidden and this code fragment is copied by cloning process to several other places without the awareness of the existence of the bug. If any instance of this propagated bug is discovered at a certain stage of evolution, its repair should take place in all code segments where it has been propagated. Thus, bug propagation by cloning causes increased modifications to the respective clones during evolution. Secondly, a new change made in a clone fragment might need to be propagated to other clones falling in the same clone family to maintain consistency. Whether such changes propagate consistently or inconsistently, there is no doubt that they increase efforts during software evolution. Thus, we see that the negative impacts of clones are directly related to higher changes in cloned code. In other words, negative impacts of clones increase software instability.

Hence, if we can identify the changes occurring in the cloned and non-cloned regions of a software system and can make a comparative analysis of these changes, we will be able to understand the impact of clones on maintenance for that software system. From this assumption we limit our study on the eight metrics (as well as studies [19], [24], [33], [36], [37], [41], [42] listed in Table I) that represent

TABLE I: Stability Measurement Metrics and their Implications

Metric	**Implication**
Modification frequency (MF) of cloned or non-cloned code [24]	*Modification frequency* is the measurement of how frequently a code region (clone or non-clone) gets modified. It focuses on the count of changes ignoring the quantity (or amount) of lines affected by a change. A high frequency of modifications in a code region indicates a high instability of that code region.
Modification probability (MP) of cloned or non-cloned code [19]	This metric is originally termed as *overall instability* by Göde and Harder [19]. *Modification probability* determines what proportion of the tokens in cloned or non-cloned code gets modified per commit operation. If a code region exhibits a high modification probability, then it indicates that the code region has a high instability.
Average last change dates (ALCD) of clone or non-clone LOC [33]	*Average last change date* determines how lately a code region (cloned or non-cloned) gets modified. According to this metric, a code region that got changed more lately is considered more unstable.
Average age (AvgAge) of clone or non-clone LOC [41]	*Average age* (slightly different from *average last change date*) calculates how long a cloned or non-cloned line of code remains unchanged on an average. According to the consideration of this metric, a source code line that remains unchanged for a longer time is considered more stable.
Impact of changes in cloned and non-cloned code [37]	Impact of changing a particular cloned or non-cloned method indicates the number of other methods that we also need to change (i.e., co-change) as a consequence of changing that particular cloned or non-cloned method. A high impact is an indicator of high instability.
Likelihood of changes in cloned and non-cloned code [37]	*Likelihood of changes* quantifies the change probability of a particular cloned or non-cloned method. A higher value of this metric for a method indicates a higher instability of that method.
Average instability per cloned method (AICM) due to cloned or non-cloned code [36]	This metric is a composite one incorporating two proportions: (i) average proportion of cloning in cloned methods (*EPCM*) and (ii) average percentage of changes to the clones in cloned methods (*CPCM*). *Average instability per cloned method* determines the average proportion of the number of changes occurring in the clone portions of cloned methods to the total number of changes in the cloned methods. In other words, this is the probability by which changes take place in the clone portions of the cloned methods. A higher value of *CPCM* compared to *EPCM* is an indicator of higher instability of cloned code compared to non-cloned code.
Change dispersion in cloned and non-cloned code [42]	*Dispersion of changes* quantifies the extent to which the changes in the clone or non-clone regions are scattered over the respective region. A higher dispersion of changes in a code region indicates a higher instability of the corresponding code region.

TABLE II: Research Questions (RQ)

Research Questions Corresponding to Eight Metrics		
	Research Questions	**Metrics and Studies**
RQ1	Which code changes more frequently, cloned or non-cloned?	Modification Frequency (MF), Hotta et al. [24]
RQ2	Which code exhibits higher modification probability, cloned or non-cloned?	Modification Probability (MP), Göde and Harder [19]
RQ3	Which code changed more recently, cloned or non-cloned?	Average Last Change Date (ALCD), Krinke [33]
RQ4	Which code remains unchanged for greater lengths of time, cloned or non-cloned?	Average Age (AA), Our study [41]
RQ5	Which method exhibits higher impact (elaborated in Section IV-E) of changes in it, cloned or non-cloned?	Impact, Lozano and Wermelinger [37]
RQ6	Which method is more likely to change, cloned or non-cloned?	Likelihood, Lozano and Wermelinger [37]
RQ7	Which code in partially cloned methods exhibits higher average instability, cloned or non-cloned?	Average Instability per Cloned Method (AICM), Lozano and Wermelinger [36]
RQ8	Which code gets more scattered changes, cloned or non-cloned?	Change Dispersion (CD), Our study [42]
Research Questions Corresponding to Drawbacks of Existing Studies		
	Research Questions	**Drawbacks**
RQ9	Do different types of clones exhibit different stability?	DES 3 (Section II-A)
RQ10	Do clones of different programming languages show different stability?	DES 4 (Section II-A)
DES = Drawback of Existing Studies		

different aspects of stability. We did not consider studies [8], [12], [27], [29], [31], [60] that aim to identify whether clones introduce bugs or are maintained consistently, or not. We implement all the stability metrics listed in Table I in a uniform framework and investigate these metrics for the cloned and non-cloned code in thousands of revisions of 12 diverse subject systems. We develop our uniform framework focusing on the commonality of experimental settings for investigating different stability metrics. We will describe this framework in Section III. According to our experimental results and analysis we summarize our findings in the following way:

- Cloned code is often (not always) more unstable than non-cloned code in the maintenance phase.
- Both Type 1 and Type 3 clones appear to be more unstable compared to Type 2 clones in the maintenance phase. We should possibly prioritize Type 1 and Type 3 clones when taking clone refactoring decisions.
- Clones in Java and C systems exhibit a higher instability than the clones in C#. We suggest that code clones in Java and C systems should be prioritized for refactoring. Clones in C# are often more stable than non-cloned code in these systems. Thus, we can possibly exclude C# systems from considerations when taking clone refactoring decisions.
- It seems that object-oriented programming languages promote more cloning compared to procedural programming languages. However, changes to the clones in procedural languages are more scattered compared to the changes to the clones in object-oriented languages.
- From our correlation analysis of the eight stability metrics we find that metrics in either of the two sets: (change Dispersion, Impact, and AICM) or (Change Dispersion, Likelihood, and AICM) can be investi-

gated in order to realize the stability scenario of code clones in a candidate software system.

We have also used our implemented uniform framework for systematically replicating the original studies. In our replication we have used the same subject systems (additional 16 subject systems for all studies), clone detection tools (additional two clone detection tools), and tool settings as of the original studies. We find that the experimental results obtained from our replication experiments are mostly equivalent to the experimental results of the original studies.

The rest of the paper is organized as follows. Section II mentions the drawbacks of the existing clone stability studies and discusses our contribution towards addressing these drawbacks, Section III describes our uniform framework for evaluating the stability metrics, Section IV elaborates on the stability measuring methodologies and metrics, Section V describes the experimental steps, Section VI discusses our subject systems, Section VII presents and analyzes our experimental results, Section VIII presents cumulative statistics and analysis of the stability metrics, Section IX clarifies our findings and provides actionable recommendations, Section X mentions the possible threats to validity, Section XI discusses the related work, and Section XII concludes the paper.

II. Motivation and Contribution

In this section we identify the lackings of the existing studies on comparing the stability of cloned and non-cloned code, and discuss our contribution towards addressing those.

A. Problem Identification

A number of stability studies [19], [24], [32], [33], [36], [37], [41] tried to identify, analyze, and compare the changes occurring in the cloned and non-cloned code of different software systems. However, these studies did not agree about the comparative stability of cloned and non-cloned code. As a result, there is no concrete answer to the long lived research question: 'Is cloned code really stable in the maintenance phase?'. To illuminate this question further, we investigated each of the prior stability-related studies. We identified the following drawbacks in the existing studies.

(1) Lack of a common framework: Different studies were conducted on different experimental setups, more specifically

- on different sets of subject systems
- using different clone-detection tools with different parameters
- on releases or different sets of revisions (of subject systems) taken at different intervals. Considering revisions at particular time-intervals has the potential to disregard a significant portion of changes that occurred to the code base during those intervals.
- inconsistent preprocessing of subject systems.

Thus, different studies may have different outcomes.

(2) Investigation on insufficient metrics: Different studies investigated different subsets of metrics. However, a complete assessment of impacts requires the assessment of all of the existing metrics on the same experimental settings.

(3) Lack of investigation on different types of clones: None of these studies except [41] could draw a clear comparison among the impacts of different clone types because the clone detection tools used in these studies cannot detect different types of clones separately. Such a comparison is very important, because this can suggest us to concentrate on clone types that are highly change-prone compared to others that are less change-prone (i.e., comparatively more stable) and can thus reduce a significant amount of refactoring efforts being spent for comparatively stable clone types.

(4) Lack of programming-language centered investigation: None of the existing studies investigated whether the same clone types in different programming languages behave in different ways and show different impacts as well. This information can help software developers to be more careful while developing projects with programming languages where clones exhibit high instability.

(5) Lack of system diversity: Most of the studies have drawn conclusions without investigating a wide variety of subject systems.

B. Our Contribution

Focusing on the above issues we perform an in-depth empirical study where we evaluate all known (eight in total) stability measurement metrics (proposed in the studies [19], [24], [33], [36], [37], [41], [42]) on the same experimental setup which we term as a uniform framework. The stability metrics and the corresponding studies have already been listed in Table I. Different metrics were calculated following different techniques. We term these techniques as methodologies in the rest of the paper. We implement these methodologies (seven methodologies in total from seven studies [19], [24], [33], [36], [37], [41], [42]) on our uniform framework and apply them on twelve subject systems of diversified sizes, application domains, purposes and implementation languages. Here, we should mention that most of the existing studies investigated only a small number of Java systems (two systems in [19], three systems in [33], five systems in each of [37] and [36]) which were not of wide variety.

We implemented the candidate methodologies for calculating the metrics using two clone detection tools: NiCad [50], CCFinderX [14]. We analyzed our experimental results from four different dimensions: (1) implementation language, (2) clone-types, (3) subject systems, and (4) clone detection tools to find the answer to the central research question 'Is cloned or non-cloned code more stable during software maintenance?'. However, we decompose this central question into eight questions corresponding to the eight metrics. These questions are mentioned in Table II. The last two questions in this table address the third and fourth drawbacks.

For answering the last two questions we defined two null hypothesis as stated below.

Null hypothesis 1 (Corresponds to RQ9): *There is no difference among the stabilities of different types of clones.*

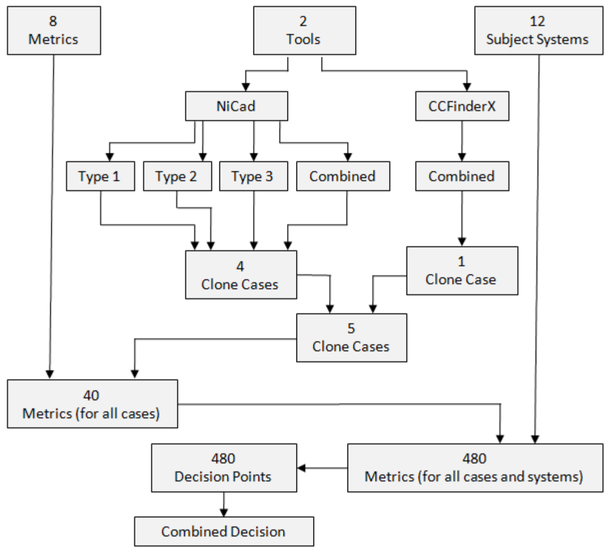


Fig. 1: The decision making procedure

Null hypothesis 2 (Corresponds to RQ10): *There is no difference among the stabilities of clones of different programming languages.*

We performed two-tailed Fisher's exact tests using the implementation at [18] on our observed data for inspecting the acceptance or rejection of these two hypotheses. We also answered each of the other eight questions with many interesting outcomes in the corresponding subsections of the result section.

III. Uniform Framework

We mentioned that different studies analyzed clone stabilities using different experimental setups which might be a potential cause to the different outcomes. Focusing on this point, we implemented all the candidate methodologies (seven in total) that calculate eight metrics (one methodology [37] calculates two metrics) on a common framework written in Java programming language using MySQL as the back-end database server. Implementation in a common framework supports time efficient sharing of intermediate results among the methodologies. We emphasized on the commonality of the data storage structure in the MySQL back-end so that once the preprocessing of files, identification of changes between files of consecutive revisions and detection of clones from each revision for a single subject system are done and outputs are stored into the database, these stored results can be used by each of the candidate methodologies to work on that subject system. Figure 1 describes our decision-making strategy based on the common framework.

The figure shows that we evaluated eight stability measurement metrics (in total). We implemented the methodologies and calculated the metrics using two clone detection tools: CCFinderX [14] and NiCad [50]. From these two clone detection tools we can obtain clone results for the following five cases.

(1) Type 1 clone results from NiCad

(2) Type 2 clone results from NiCad

(3) Type 3 clone results from NiCad

(4) Combined clone results (clone results combining above three clone types) from NiCad

(5) Clone results from CCFinderX. CCFinderX detects Type 1 and Type 2 clones in a combined way. It cannot detect Type 3 clones. Also, it cannot report code clones by separating them by clone-types.

We will explain these cases in detail in Section VII. For each of these five cases we calculate each of the eight metrics. Thus, we have 40 stability-related metrics (5 clone cases × 8 metrics) in total. We calculate these 40 metrics from each of the 12 candidate subject systems. Thus, for all subject systems, we calculated 480 (40 metrics × 12 subject systems) results in total. However, for calculating these 480 results we conducted 420 separate experiments (7 methodologies × 5 clone cases × 12 subject systems). One methodology proposed by Lozano and Wermelinger [37] calculates two metrics (Impact and Likelihood). We define each of the 480 results as a decision point from which we decide about whether cloned code is more stable than non-cloned code or not. After analyzing all of these 480 decision points from different perspectives we take a combined decision on the comparative stability of cloned and non-cloned code.

IV. Stability Measuring Methodologies and Metrics

We discuss the candidate stability measurement methodologies (seven methodologies) and related metrics (eight metrics) in the following subsections.

A. *Modification Frequency (MF) Proposed by Hotta et al.*

Hotta et al. [24] calculated: (i) MF_c (Modification Frequencies of Cloned Code or Duplicate code) and (ii) MF_n (Modification Frequencies of Non-Duplicate code) considering all the revisions of a particular code-base extracted from subversion repository. Their metric calculation strategy involves several sequential steps including: (1) identification and checking out of relevant revisions of a subject system, (2) normalization of source files by removing blank lines, comments and indents, (3) detection and storing of each line of duplicate code into the database. The differences between consecutive revisions were also identified and stored in the database. Then, MC_c (Modification Count in Duplicate code region) and MC_n (Modification Count in Non-Duplicate code region) were determined exploiting the information saved in the database and finally MF_c and MF_n were calculated using the following equations [24]:

$$MF_c = \frac{\sum_{r \epsilon R} MC_c(r)}{|R|} * \frac{\sum_{r \epsilon R} LOC(r)}{\sum_{r \epsilon R} LOC_c(r)} \quad (1)$$

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

Here, R is the number of revisions of the candidate subject system. $MC_c(r)$ and $MC_n(r)$ are the number of modifications (defined in the next paragraph) in the cloned and non-cloned code regions respectively between revisions r and

$(r+1)$. MF_c and MF_n are the modification frequencies of the cloned and non-cloned code regions of the system. $LOC(r)$ is the number of LOC in revision r. $LOC_c(r)$ and $LOC_n(r)$ are respectively the numbers of cloned and non-cloned LOCs in revision r.

According to the definition of Hotta et al. [24], a modification can affect multiple consecutive lines. Suppose, L lines of a method (or any other program entity) were modified through additions, deletions or changes. If these L lines are consecutive then, the count of modification is one. If these L lines are not consecutive then, the count of modifications equals to the number of unchanged portions within these L lines plus one.

They performed their empirical study on 15 open source subject systems incorporating the clone detectors: CCFinderX [14], Simian [59], and Scorpio [57] with a general conclusion that cloned code is modified less frequently than non-cloned code. Thus according to this study cloned code is more stable than non-cloned code.

B. Modification Probability (MP) proposed by Göde and Harder

Göde and Harder [19] replicated, extended and validated Krinke's study [32] to determine whether clones are responsible for increasing maintenance effort and if does so, to what extent. In this study, they used an incremental token-based clone detection tool. They calculated the modification probabilities of cloned and non-cloned code with respect to addition, deletion and modification according to the following equations.

Modification probability of cloned code (MP_c) was calculated using the following equation.

$$MP_c = \frac{\sum_{r\epsilon R} A_c(r) + D_c(r) + C_c(r)}{\sum_{r\epsilon R} Tok_c(r)} \qquad (3)$$

Modification probability of non-cloned code (MP_n) was determined using the equation given below.

$$MP_n = \frac{\sum_{r\epsilon R} A_n(r) + D_n(r) + C_n(r)}{\sum_{r\epsilon R} Tok_n(r)} \qquad (4)$$

In the above equations, R is the set of all revisions of the candidate subject system. $A_c(r)$, $D_c(r)$, and $C_c(r)$ are respectively the total number of tokens added, deleted and changed (or modified) in the cloned regions of revision r of the subject system. In the same way, $A_n(r)$, $D_n(r)$, and $C_n(r)$ are the total number of tokens added, deleted and modified in the non-cloned regions of revision r. $Tok_c(r)$ and $Tok_n(r)$ are the total number of tokens in respectively the cloned and non-cloned regions of the subject system.

However, the modification probabilities MP_c and MP_n were termed as overall instability of cloned and non-cloned code in the original study [19]. We named these as modification probabilities considering the equations Eq. 4.3 and Eq. 4.4. The right sides of these equations determine the ratios of the modified tokens to the total number of tokens in cloned and non-cloned regions respectively.

Göde and Harder performed this study [19] on two open source Java systems with a general conclusion that cloned code is more stable than the non-cloned code.

C. Average Last Change Date (ALCD) Proposed by Krinke

Krinke [33] introduced a new concept of code stability measurement by calculating the average last change dates of cloned and non-cloned regions of a code-base based on the *blame* annotation in the SVN repository. He considers only a single revision (generally the last revision) (Hotta et al. [24] considers all the revisions up to the last one as is already described). He calculates the average last change dates of cloned ($ALCD_c$) and non-cloned ($ALCD_n$) code from the file level and system level granularities in the following way.

File level metrics

- Percentage of files where the average last change date of cloned code is older than that of non-cloned code (cloned code is older than non-cloned code) in the last revision of a subject system.
- Percentage of files where the average last change date of cloned code is newer than that of non-cloned code (cloned code is younger than non-cloned code) in the last revision of a subject system.

System level metrics

- Average last change date of cloned code ($ALCD_c$) for the last revision of a candidate subject system.
- Average last change date of non-cloned code ($ALCD_n$) for the last revision of a candidate subject system.

Calculation of average last change date (ALCD): Krinke calculated the average last change dates in the following way. Suppose five lines in a file correspond to five revision 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. We see that this process of calculating ALCD has the possibility of introducing rounding error. In the given example, the fraction '0.25' cannot be reflected to the calculated ALCD. This might force the ALCDs of cloned and non-cloned code to be equal [41].

According to the average last change date calculation process described above, we calculate the average last change date of cloned ($ALCD_c$) and non-cloned ($ALCD_n$) code in the following ways.

$$ALCD_c = ODCL + \frac{\sum_{l\epsilon CL} (DATE(l) - ODCL)}{|CL|} \qquad (5)$$

$$ALCD_n = ODNL + \frac{\sum_{l\epsilon NL} (DATE(l) - ODNL)}{|NL|} \qquad (6)$$

In the above equations (Eq. 5 and Eq. 6), $ODCL$ and $ODNL$ are respectively the oldest change dates of cloned and non-cloned lines in the last revision of the candidate subject system. CL and NL are the sets of cloned and non-cloned lines in the last revision. $DATE(l)$ is the change date corresponding to the source code line l.

The two ratios in file level metrics were calculated considering only the analyzable files in the last revision. The set of analyzable files consists of those files which contain both cloned and non-cloned code. The files containing no cloned code and the fully cloned files were excluded from consideration while determining file level metrics. As these files contain only cloned or only non-cloned code, none of these files can determine whether cloned or non-cloned code is older. File level metric is useful to determine what percentage of files takes part in defining the stability scenario of cloned and non-cloned code for a subject system. However, analyzing this percentage is not the goal of our research. Our aim is to compare the stability of cloned and non-cloned code. For this reason, in our experiment we calculated the system level metrics only. System level metrics are calculated considering all source files in a code-base. The intuition behind this methodology is that the older the code is the more stable it is. That means, if a code region (cloned or non-cloned) remains unchanged for longer duration compared to the other (non-cloned or cloned) the former code region can be regarded as more stable.

Krinke performed this study on three open source Java systems using Simian [59] clone detector considering only Type 1 clones with the conclusion that cloned code is more stable than non-cloned code.

D. Average Age (AA) proposed by Mondal et al.

We have just described Krinke's methodology [33] for calculating the average last change date of cloned and non-cloned lines of a code-base. The outputs of this methodology are dates. A variant of this methodology proposed by Mondal et al. [41] analyzes the longevity (stability) of cloned and non-cloned code by calculating their average ages (in days). This methodology also uses the *blame* command of SVN (as was used by Krinke [33]) to calculate the age for each of the cloned and non-cloned lines in a subject system.

1) Average Age measurement technique: Suppose we have several subject systems. For a specific subject system this methodology works on the last revision r_o. By applying a clone detector on revision r_o, the lines of each source file can be separated into two disjoint sets: (i) one containing all cloned lines and (ii) the other containing all non-cloned lines. Different lines of a file contained in r_o can belong to different previous revisions. If the *blame* command on a file assigns the revision r to a line x, then it is understood that line x was produced in revision r and has not been changed up to last revision r_o. The creation date of r is denoted as $DATE(r)$. In the current revision R, the age (in days) of this line is calculated by the following equation:

$$AGE(x) = DATE(r_o) - DATE(r) \tag{7}$$

Two average ages of cloned and non-cloned code were calculated from system level granularity as follows.

- 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. The equation for calculating AA_c is given below.

$$AA_c = \frac{\sum_{l \epsilon CL} (LRD - DATE(l))}{|CL|} \tag{8}$$

  In the above equation, CL is the set of all cloned lines in the last revision of the candidate subject system. LRD is the creation date of the last revision of the subject system. $DATE(l)$ is the last change date of source code line l.

- 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 in the following way.

$$AA_n = \frac{\sum_{l \epsilon NL} (LRD - DATE(l))}{|NL|} \tag{9}$$

  In the above equation, NL is the set of all non-cloned lines in the last revision of the candidate subject system.

According to this methodology, a higher average age is the implication of higher stability. The study [41] using this methodology was conducted on twelve open-source subject systems with the general conclusion that cloned code exhibits higher instability than non-cloned code. Such a finding is contradictory to the findings of the previously described studies.

E. Likelihood and Impact of Methods proposed by Lozano and Wermelinger

Lozano and Wermelinger performed an in-depth study [37] to assess the impacts of clones on maintenance by examining all the revisions of candidate subject systems. Their calculations were based on method level granularity using CCFinderX [14] clone detection tool. According to their definition, a cloned method can be fully or partially cloned (only a portion of the method is cloned). They detected the methods in different versions using CTAGS [17]. They also performed the origin analysis of methods in consecutive revisions to see how a method gets changed as it passes through multiple revisions. Using origin analysis they separated the methods into the following three subsets.

Always cloned methods: An always cloned method contains a cloned portion in it in all revisions in which it remains alive.

Never cloned methods: A never cloned method contains no cloned portions in its total lifetime.

Sometimes cloned methods: A sometimes cloned method contains cloned portions for a limited period of its lifetime.

They calculated the following stability metrics.

Likelihood: The likelihood of change of a method m during its cloned period (or non-cloned period) is the ratio between the number of changes to m and the total number of changes to the system (all methods) during cloned period (or non-cloned period).

$$ICC = \frac{\sum_{m \epsilon M_c} Impact_{cloned}(m) + \sum_{m \epsilon M_{sc}} Impact_{cloned}(m)}{|M_c| + |M_{sc}|} \tag{10}$$

$$Impact_{cloned}(m) = \frac{\sum_{c \epsilon CCP(m)} CCM(c)}{|CCP(m)|} \tag{11}$$

$$INC = \frac{\sum_{m \epsilon M_n} Impact_{non-cloned}(m) + \sum_{m \epsilon M_{sc}} Impact_{non-cloned}(m)}{|M_n| + |M_{sc}|} \tag{12}$$

$$Impact_{non-cloned}(m) = \frac{\sum_{c \epsilon CNP(m)} CCM(c)}{|CNP(m)|} \tag{13}$$

$$LCC = \frac{\sum_{m \epsilon M_c} Likelihood_{cloned}(m) + \sum_{m \epsilon M_{sc}} Likelihood_{cloned}(m)}{|M_c| + |M_{sc}|} \tag{14}$$

$$Likelihood_{cloned}(m) = \frac{\sum_{c \epsilon CCP(m)} NCM(m, c)}{\sum_{c \epsilon CCP(m)} \sum_{m_i \epsilon M} NCM(m_i, c)} \tag{15}$$

$$LNC = \frac{\sum_{m \epsilon M_n} Likelihood_{non-cloned}(m) + \sum_{m \epsilon M_{sc}} Likelihood_{non-cloned}(m)}{|M_n| + |M_{sc}|} \tag{16}$$

$$Likelihood_{non-cloned}(m) = \frac{\sum_{c \epsilon CNP(m)} NCM(m, c)}{\sum_{c \epsilon CNP(m)} \sum_{m_i \epsilon M} NCM(m_i, c)} \tag{17}$$

Impact: The impact of a method m is denoted by the average percentage of the system that gets changed whenever m changes during its cloned period or non-cloned period. This is calculated by the average number of methods changed on those commits where method m gets changed.

We calculate the average impact of always cloned methods and the cloned periods of sometimes cloned methods. We term this impact as the impact of cloned code (ICC). We calculate *ICC* according to Eqs. 10 and 11.

In the equations (Eq. 10 and Eq. 11), M_c, M_{sc}, and M_n are the sets of always cloned, sometimes cloned and never cloned methods. $Impact_{cloned}(m)$ is the impact of the cloned period of method m. $CCP(m)$ is the set of all commits where m was changed during its cloned period. $CCM(c)$ is the count of methods changed in commit c.

We also calculate the average impact of never cloned methods and the non-cloned periods of sometimes cloned methods. We term this impact as the impact of non-cloned code (INC). INC is calculated according to Eqs.12 and 13.

In the equations (Eq.12 and Eq. 13), $Impact_{non-cloned}(m)$ is the impact of the non-cloned period of method m. $CNP(m)$ is the set of all commits where m was changed during its non-cloned period. Other terms are already described for the equations: Eq. 10 and Eq. 11. We also calculate the likelihood of cloned (LCC) and non-cloned code (LNC). LCC is the average likelihood considering always cloned methods and the cloned periods of the sometimes cloned methods. We calculate LCC according to the equations - Eq. 14 and Eq. 15.

In the equations (Eq. 14 and Eq. 15), $Likelihood_{cloned}(m)$ is the likelihood of the method m during its cloned period. $NCM(m, c)$ is the number of changes to the method m in commit c. M is the set of all methods. The remaining terms in these equations have already been defined. We also calculate LNC, the average likelihood considering never cloned methods and the non-cloned periods of the sometimes cloned methods, according to the equations - Eq. 16 and Eq. 17.

In Eq. 17, $Likelihood_{non-cloned}(m)$ is the likelihood of the method m during its non-cloned period.

Lozano and Wermelinger performed this study [37] on five open source Java systems with a conclusion that cloned methods are more likely to be modified compared to the non-cloned methods. Also, cloned methods seem to increase maintenance efforts considerably.

F. Average instability per cloned method (AICM) proposed by Lozano and Wermelinger

The methodology proposed in [36] is an improvement of the previous work [37] and gives more sophisticated analysis of the impacts of clones. In the previous study [37] several issues such as clone families, inclusion and exclusion of clones in families as well as late propagation of changes were not considered. But, in this study [36] all these matters were taken into account and many more measurements were made using the same tools. The implementation and analysis are based on method-level granularity where the definition of cloned method remains the same as described in the previous methodology [37]. We measured the following two metrics from this methodology, because these are related to code stability. These two metrics together can help us to determine whether cloned or non-cloned code is more stable. We term these two metrics together as *average instability per cloned method (AICM)* (as is mentioned before).

Extension per cloned method (EPCM): This metric was calculated by determining the average proportion of cloned tokens in the cloned methods at a particular commit.

Proportion of changes to the clones of cloned methods (CPCM): We calculated this metric by the average ratio between the number of changes in the cloned tokens and the total number of changes in the cloned methods up to a particular commit. This metric is originally termed as *Stability per method* by Lozano and Wermelinger [36].

For a particular commit operation we determine $EPCM$ and $CPCM$ according to the following equations.

$$EPCM = \frac{\sum_{m \epsilon M} \frac{CTok(m) \times 100}{Tok(m)}}{|M|} \tag{18}$$

$$CPCM = \frac{\sum_{m \epsilon M} \frac{CTok_{changed}(m) \times 100}{Tok_{changed}(m)}}{|M|} \tag{19}$$

In the above equations (Eq. 18 and Eq. 19), M is the set of all cloned methods in a particular commit operation. $CTok(m)$ is the number of cloned tokens in a cloned method m and $Tok(m)$ is the number of total tokens in m. $CTok_{changed}(m)$ is the number of cloned tokens changed in the cloned method m. $Tok_{changed}(m)$ is the number of total tokens changed in m.

Lozano and Wermelinger performed this study [36] on five open source subject systems written in Java. According to their experimental results, cloned methods have a higher density of changes compared to the non-cloned methods. This outcome also contradicts with the conclusions drawn by Krinke [33] and Hotta et al. [24]. According to the explanation of Lozano and Wermelinger, such a contradiction occurred because of different setups of the different clone detection tools incorporated in these studies.

G. Dispersion of Changes (CD)

Mondal et al. [42] proposed and implemented a methodology for measuring the dispersions of changes in the cloned and non-cloned regions considering method level granularity. According to their definition, *change dispersion* in a particular code region (cloned or non-cloned) is the percentage of method genealogies affected by changes in that region during that particular period of evolution. As the evolution period they considered the total duration starting from the first revision up to the creation of the last revision of a particular software system. Each of the commit operations in the evolution period involving some modifications to the source code was taken into account. *Change dispersion* is calculated in the following way.

Calculation of Dispersion: A method (or function in case of C language) is defined as a cloned method when it contains some cloned lines in it. According to this methodology [42] there are two types of cloned methods: (i) fully cloned methods (all of the lines contained in these methods are cloned lines) and (ii) partially cloned methods (these methods contain some non-cloned portions in them). For calculating the dispersion of cloned code, the changes in the cloned portions of the cloned (fully or partially) methods are considered. Partially cloned methods are also considered while calculating the dispersion of non-cloned code because, changes might occur in the non-cloned portions of the partially cloned methods. Moreover, while determining method genealogies it might be seen that a partially cloned method has become fully cloned or fully non-cloned after receiving a change. These methods are considered in calculating the dispersions of both cloned and non-cloned code.

Suppose, for a subject system, the sets of cloned and non-cloned method genealogies are C and N respectively. C_c is the set of cloned method genealogies which received some changes in their cloned portions during the evolution. The number of changes received by the genealogies in the set C_c is generally greater than $|C_c|$, because a particular method genealogy can experience multiple changes during evolution. In the same way, N_c is the set of non-cloned method genealogies that received some changes in their non-cloned portions. The dispersion of changes in cloned code (CD_c) and non-cloned code (CD_n) were expressed by the following equations.

$$CD_c = \frac{|C_c| \times 100}{|C|} \tag{20}$$

$$CD_n = \frac{|N_c| \times 100}{|N|} \tag{21}$$

Implementation of this methodology requires the extraction of method genealogies. We extracted method genealogies using the algorithm proposed by Lozano and Wermelinger [37]. According to the study [42], cloned code generally shows higher change dispersion compared to non-cloned code. Thus, cloned code is expected to require higher maintenance effort than non-cloned code.

V. Experimental Steps

We implemented the seven (all known) stability measurement methodologies described in the previous section using two clone detection tools for conducting this experiment. While three of these methodologies [36], [37], [42] calculate the respective stability metrics considering method level granularity, the remaining four methodologies [19], [24], [33], [41] calculate using block granularity. In the following subsections we describe the sequential steps for calculating the candidate metrics.

A. Extraction of Repositories.

All of the subject systems on which we applied our candidate methodologies were downloaded from open-source SVN repositories. For each system, we extracted only those revisions which were created because of some source code modification (addition, deletion or change) rather than just tagging and branching operations. To determine whether a revision should be extracted or not, we checked the extensions of the files which were modified to create the revision. If some of these modified files are source files, we considered the revision as our target revision and extracted it.

B. Preprocessing.

We applied the following two preprocessing steps to all the revisions of the subject systems before applying the methodologies on them except for Krinke's methodology [33] and average age calculation process [41].

(1) Rearrangements of lines so that an isolated left or right brace (if a left or right brace remains in a line associated with no other character) was deleted and added at the end of the previous line.

(2) Deletion of blank lines and comments.

These preprocessing steps were done to avoid the effects of changes to the comments and indentations on the calculated metrics.

It was not possible to apply the mentioned preprocessing steps in case of Krinke's methodology [33] and its variant because these work on the output of SVN $blame$ command for a specific file, not on the original file. For these methodologies (Krinke's methodology [33], and the average age calculation methodology [41]), we just ignored the blank lines and comments from $blame$ command output.

C. Detection of method.

In order to detect the methods of a specific revision we applied CTAGS [17] on the source files of that revision. For each method we determined its (1) file name, (2) class name (Java and C# systems), (3) package name (Java), (4) method name, (5) signature, (6) starting and ending line numbers, and (7) revision number

In case of the subject systems written in C, we determined five properties from the above list excluding class name and package name. We detected the methods for all target revisions. The method detection process for a particular revision can be made faster by reusing the methods stored for the immediate previous revision. If we have completed the detection and storage activities for revision r_i, we do not need to apply CTAGS [17] for the source files which remain unchanged in revision r_{i+1}. Methods of these unchanged files can be retrieved from the database and forwarded to the r_{i+1}. We apply CTAGS [17] only to those source files which received some changes while being forwarded from r_i to r_{i+1}. However, we do not store the methods in the database at this stage because we need the information about which methods have clones and which methods have got changed before being forwarded to the next revision.

D. Clone Detection

As noted previously, we applied NiCad [50] and CCFinderX [14] clone detection tools to each target revision to detect clone blocks. These clone blocks were then mapped to the already detected methods of this revision by comparing the beginning and ending line numbers of clone blocks and methods. So, for each method we collect the beginning and ending cloned line numbers (if exist). CCFinderX currently outputs the beginning and ending token numbers of clone blocks. We automatically retrieve the corresponding line numbers from the generated preprocessed files. We store the clones as well as clone families detected from a revision in the database. We used the following setups for the clone detection tools.

Setup for CCFinderX: CCFinderX [14] is a token based clone detection tool that currently detects block clones of Type-1 and Type-2. We set CCFinderX to detect clone blocks of minimum 30 tokens with TKS (minimum number of distinct types of tokens) set to 12 (as default).

TABLE III: NiCad Settings

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

Setup for NiCad: NiCad [16] 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. We used the NiCad settings in Table III for detecting three types of clones. 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. These settings of NiCad are considered standard [52]–[55] and for all these settings NiCad was shown to have high precision and recall [51], [52]. Also, the settings that we have used for CCFinderX are considered equivalent to those of NiCad [39].

E. Detection and Reflection of Changes

We identified the changes between corresponding files of consecutive revisions using UNIX *diff* command. *diff* outputs three types of changes: (1) addition, (2) deletion, and (3) modification

with corresponding line numbers. We mapped these changes to methods using line information. So, for each method we gathered two more pieces of information: (1) the count of lines changed in cloned portions and (2) the count of changed lines in non-cloned portions. We map the changes to methods, and also store the changes in the database with corresponding line information.

F. Storage of Methods

At this stage, we have all necessary pieces of information of all methods belonging to a particular revision. We store these methods in database with individual entry for each method. The attributes that we store for a particular method have already been listed in Section V-C. We do not store the statements inside a method. For each of the revisions of a particular software system we detected the methods along with cloning and change information and stored the methods in the database. So, our database contains different method sets for different revisions: one set for each revision.

G. Method Genealogy Detection

After completing method detection and storage for all revisions of a subject system, we detected method genealogies following the technique proposed by Lozano and Wermelinger [37] to identify the propagation of methods across revisions. Suppose a method was created in revision r_i and was alive and propagated to the next two revisions r_{i+1} and r_{i+2} with or without some changes. So, this method has corresponding entries in all of these three revisions. By detecting method genealogies we can identify these entries as belonging to the same method. For detecting genealogies we assign a single

TABLE IV: Subject Systems

	Systems	Domains	LLR	Revisions
Java	DNSJava	DNS protocol	23,373	1635
	Ant-Contrib	Web Server	12,621	176
	Carol	Game	25,092	1699
	jabref	Reference Management	59,648	3000
C	Ctags	Code Definition Generator	33,270	774
	Camellia	Multimedia	100,891	608
	QMail Admin	Mail Management	4,054	317
	GNUMake Uniproc	Auto-build system for C/C++ projects	75,745	863
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

LLR = LOC in Last Revision

unique ID to all of the entries of a particular method residing in different revisions.

H. Metrics Calculation

After completing all the steps described above for a particular subject we calculated the metrics. We calculate four metrics: (1) impact, (2) likelihood, (3) AICM (Average Instability per Cloned Method), and (4) CD (Change Dispersion) using the method information stored in the database. The remaining four metrics: (5) MF (Modification Frequency), (6) MP (Modification Probability), (7) ALCD (Average Last Change Date), and (8) AA (Average Age) are calculated using the information stored for clones and changes.

VI. Subject Systems

Table IV lists the subject systems that were included in our study along with their associated attributes. We downloaded the subject systems from SourceForge[1]. The subject systems are of diverse variety in terms of the followings.

(1) **Application domains:** The candidate systems span 10 different application domains as mentioned in Table IV.

(2) **Implementation language:** The systems cover three programming languages: Java, C, and C#.

(3) **System size:** The systems are of different sizes, from very small (4 KLOC) to large (100 KLOC).

We have also systematically replicated the original studies. For the purpose of replication with original setups, we downloaded the subject systems used in the original studies and carried out the same analysis steps as in the original studies. We analyzed 16 additional subject systems written in Java, C and C++ of various sizes, application domains and lengths of revision history. The details of the replication experiments are presented in Section VII-H.

VII. Experimental Results and Analysis

In this section, we presented the results obtained for eight stability metrics in eight tables. The corresponding analysis for each of these metrics (as well as table data) is also given in this section.

[1]SourceForge: http://sourceforge.net/

Normalization of metric values: The values corresponding to five candidate metrics: *modification probability*, *impact*, *likelihood*, *change dispersion*, and *average instability per cloned method* (*EPCM*, *CPCM*) are normalized within the range zero to one. The equations for calculating *modification probabilities* (Eq. 3, Eq. 4), *impacts* (Eq. 10, Eq. 12), and *likelihoods* (Eq. 14, Eq. 16) provide us values which are normalized within zero to one. However, the equations for calculating *change dispersions* (Eq. 20, Eq. 21), *EPCMs* (Eq. 18), and *CPCMs* (Eq. 19) give us percentages. We normalize these percentages within zero to one by dividing them by 100. We performed this normalization because we wanted the values of the metrics in the same range.

However, it was not possible to normalize the values of the remaining three metrics: *modification frequency*, *average last change date*, and *average age* in a particular range (e.g. zero to one) because of the following reasons.

(1) None of these metrics has a fixed upper bound.

(2) The values corresponding to the metrics *average last change date* and *average age* are dates and ages (in days) respectively.

The experimental results corresponding to each metric can be broadly divided into two categories based on the discrimination power of the clone detectors:

(1) Individual type results: These results assist us to analyze the influence of each type of clones on the stability measurement metrics individually. Individual type results were obtained by applying NiCad clone detector, because it not only detects the three major types (Type 1, Type 2 and Type 3) of clones in a combined way but also facilitates the separation of three types clones from one another. NiCad detects clones by separating them into individual classes. Generally, Type 2 clone detection results of NiCad include Type 1 clone classes. In the same way, the results obtained by detecting Type 3 clones include both Type 1 and Type 2 clone classes. To get the exact Type 2 clone classes we excluded Type 1 classes from Type 2 results. In the same way, exact Type 3 clone classes were obtained by excluding Type 2 and Type 1 classes from Type 3 results. However, separation of individual clone types is not possible with CCFinderX.

(2) Combined type results: These results help us to analyze the combined effect of three types of clones on the stability measurement metrics. We used both NiCad and CCFinderX to get these results. For determining the combined impacts of three types of clones using NiCad, we used the Type 3 clone detection results without excluding Type 1 and Type 2 classes. We also used CCFinderX to get combined type results. However, CCFinderX detects only Type 1 and Type 2 clones.

For each metric we present a table containing both individual type and combined type results. We interpreted the tables as consisting of *decision points* as was done by Mondal et al. [39].

Decision point: A particular decision point consists of the followings.

(1) Metric value for cloned code,

(2) Metric value for non-cloned code, and

(3) A remark indicating the comparative stability of cloned and non-cloned code considering that metric value.

The decision points regarding the individual type results reflect the stability scenario of three types (Type 1, Type 2, Type 3) of clones. The overall stability of cloned code combining three clone-types is reflected by the decision points regarding the combined type results.

Mondal et al. [39] categorized the decision points into three categories by calculating an *eligibility value* for each decision point and by determining whether the eligibility value exceeds a threshold value or not. They considered a threshold value of 10 in their research work. In this research work, we also categorize the decision points in the same way considering this threshold value. The categories of the decision points are described below.

- **Category 1 (CLONES MORE STABLE):** For each of the points belonging to Category 1, two conditions hold: (1) the eligibility value is greater than the threshold value, and (2) cloned code appears to be more stable than non-cloned code. These points are marked with $\oplus$.
- **Category 2 (CLONES LESS STABLE):** For each of the points belonging to this category, two conditions hold: (1) the eligibility value is greater than the threshold value, and (2) cloned code appears to be less stable than non-cloned code. These points are marked with $\ominus$.
- **Category 3 (NEUTRAL):** For each of the points belonging to this category, the eligibility value is less than the threshold value. Such a point is regarded as an insignificant decision point and is marked with $\bigcirc$.

Eligibility Value: The following equation calculates eligibility value for each decision point.

$$Eligibility\ Value = \frac{(HMV - LMV) * 100}{LMV} \qquad (22)$$

Here, HMV is the higher value of the metric values corresponding to cloned and non-cloned code for a particular decision point. LMV is the smaller one of the two metric values corresponding to cloned and non-cloned code for that decision point. If this *eligibility value* corresponding to a particular decision point is greater than or equal to a threshold value 10, as was selected by our previous study [42], we say that the decision point is significant and falls in Category 1 or Category 2. We select the threshold magnitude of *eligibility value* in such a way that, it will force a decision point having larger but very near metric-values (such as 41 and 40. *Eligibility Value* = (41-40)*100/40 = 2.5) to be selected as insignificant while a decision point with smaller but near metric-values (such as 3 and 4. *Eligibility Value* = 33.33) to be selected as significant which is expected.

We analyze the comparative stability of cloned and non-cloned code considering the significant decision points (belonging to Category 1 and Category 2). We do not consider the insignificant points (belonging to Category 3) because in case of such a decision point the two stability metric values regarding the cloned and non-cloned code are almost the same (i.e., the difference between the values is negligible). In our experiment, the amount of insignificant decision points is very low compared to the significant decision points. Considering all the eight metrics, we have 480 decision points in total. While only 46 of these points belong to Category 3 (i.e., insignificant decision points), respectively 196 and 238 points belong to Category 1 and Category 2.

The eligibility value calculation technique [39] could not be applied for the decision points corresponding to the metrics: *average last change date (ALCD)*, *average age (AA)*, and *average instability per cloned method (AICM)*. We cannot calculate eligibility values for these exceptions, because: (1) dates are ordinal, not cardinal, so any difference is significant, (2) average age and AICM require detailed analysis (Sections VII-D and VII-F).

We analyzed the results obtained for each metric individually from the following four perspectives.

(1) Overall analysis: This analysis is based on the combined type results. Considering the combined type results from each clone detector (NiCad, CCFinderX) in each table we calculated three proportions: (i) the proportion of decision points belonging to Category 1 (CLONES MORE STABLE), (ii) the proportion of the decision points belonging to Category 2 (CLONES LESS STABLE), and (iii) the proportion of the decision points belonging to Category 3 (NEUTRAL). We show these proportions in a bar-graph and provide our analysis on these. The following example will explain this.

Example: We first consider the 12 decision points in Table V under the heading, **NiCad-Combined**. We see that 4 points (33.33% of the 12 decision points) are insignificant (belong to Category 3). According to 3 decision points (i.e., 25% of the 12 decision points under **NiCad-Combined** category), cloned code appear to be less stable than non-cloned code. The opposite is true for remaining 5 decision points (41.67% of 12 decision points). Disregarding the insignificant points we can say that according to NiCad clone detector, cloned code is generally less frequently modified compared to non-cloned code. In the same way, we also analyze the 12 decision points under **CCFinderX-Combined** category. Fig. 2 shows a bar-graph that indicates the percentages of different categories of decision points considering **NiCad-Combined** results and **CCFinderX-Combined** results separately.

(2) Programming-language centric analysis: We compare the stabilities of cloned and non-cloned code with respect to the three programming languages in language centric analysis. Here, we also identify which language exhibits higher instability of clones according to which stability metric. This analysis is based on combined type results. In each table for a particular metric, we get four decision points under the combined-type category of each clone detector. Considering these four decision points, we determine the proportion of decision points belonging to Category 1, Category 2, and Category 3. We present a graph for each metric showing the proportions regarding each clone detection tool (NiCad, and CCFinderX).

Example: As an example, all the 4 points belonging to Java in Table V under the heading **CCFinderX-Combined** are significant. Among these significant points, 3 points (75% of the points) belong to Category 2 (CLONES LESS STABLE) and 1 point (25% of the points) belongs to Category 1 (CLONES MORE STABLE). If we consider the 4 decision points for Java under the **NiCad-Combined** heading, we see that the percentages of decision points under the three categories: Category 1, Category 2, and Category 3 are 0% (no decision points under Category 1), 25% (one decision point under Category 2), and 75% (3 insignificant decision points). Fig. 3 shows the language centric analysis result for Table V.

(3) Type-centric analysis: Our type-centric analysis for each metric is based on the individual-type results (denoted by Type 1, Type 2, and Type 3). Each table for a particular metric contains 12 decision points belonging to a particular clone-type. We calculate the proportions of decision points belonging to three categories: Category 1, Category 2, and Category 3 considering these points. The following example will explain this.

Example: Table V contains the individual type results (under the headings **Type 1**, **Type 2**, and **Type 3**) obtained by applying Hotta et al.'s methodology on twelve subject systems. If we consider the decision points belonging to Type 1, we see that 8 points belong to Category 1 (CLONES MORE STABLE), 3 points belong to Category 2 (CLONES LESS STABLE), and the remaining one point belongs to Category 3 (NEUTRAL). We see that while 66.7% (8 x 100 / (8+3 + 1) = 66.7) of these points belong to Category 1, only 25% and 8.3% points belong to Category 2 and 3 respectively. Fig. 4 shows the type centric analysis result for modification frequency.

(4) Type-centric analysis for each language: This analysis is also based on the individual type results. In the table corresponding to a metric, four points belong to a particular clone type (Type 1, Type 2, or Type 3) of a particular language. From these four points we determine the proportions of decision points belonging to Category 1, 2, and 3. For each metric we draw a graph showing the type-centric analysis result for each language. The following example will explain this.

Example: Among the four points belonging to Type 3 case of Java in Table V: (i) two points belong to Category 1, (ii) one point belongs to Category 2, and (iii) one point belongs to Category 3. So, the percentages of the points belonging to Category 1, 2, and 3 are 50% (2 x 100 / 4) = 50), 25%, and 25% respectively. Fig. 5 shows the percentages for every combination of clone-type and language for this table.

However, we see that for a single metric, our type-centric analysis for a particular programming language depends only on four decision points (for each combination of language and clone type). We present a cumulative analysis in Section VIII-C considering eight metrics. In this analysis, we take language-wise type-centric decisions considering 32 decision points obtained from all combinations of languages and clone-types.

We also present cumulative analyses from different dimensions in Section VIII considering the results obtained for all the metrics. In the following subsections we present our four-dimensional analysis of the experimental results obtained for each of the eight metrics.

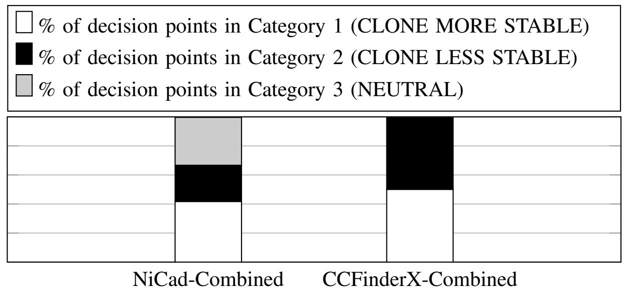


Fig. 2: Overall analysis for modification frequency.

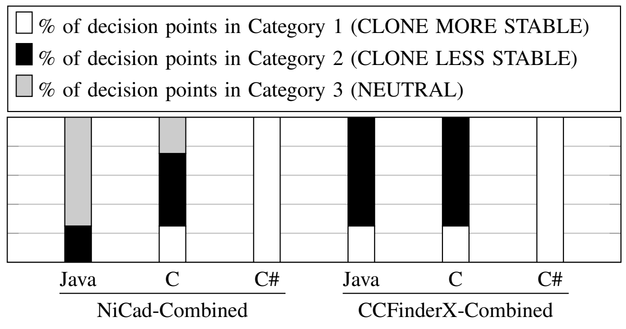


Fig. 3: Language centric statistics for modification frequency

A. *Analysis of the Experimental Results Regarding Modification Frequency*

Hotta et al. [24] calculated the modification frequencies of cloned (MF_c) and non-cloned (MF_n) code according to Eq. 1 and Eq. 2 and argued that cloned code changes less frequently than non-cloned code in general. Using our implementations of Hotta et al.'s methodology (using CCFinderX and NiCad) we calculated the modification frequencies of cloned and non-cloned code of each of the subject systems and populated the Table V. For a particular significant decision point in this table,

(i) if $MF_c < MF_n$, then changes to the cloned code are less frequent compared to the changes to non-cloned code and this point falls in Category 1 (CLONES MORE STABLE)

(ii) if $MF_c > MF_n$, then changes to the cloned code are more frequent compared to the changes to non-cloned code and this point falls in Category 2 (CLONES LESS STABLE).

The following four sections of analysis answer the first research question RQ1 from four directions.

Overall analysis: Considering the 12 decision points under the heading **NiCad-Combined** in Table V, we see that: (i) 41.67% of the points (5 points) belong to Category 1 (CLONE MORE STABLE), (ii) 25% of the decision points (3 points) belong to Category 2 (CLONE LESS STABLE), and (iii) the remaining 33% of the points (4 points) belong to Category 3 (NEUTRAL). Thus, according to NiCad clone detector, code clones are generally less frequently modified compared to non-cloned code. Such a finding confirms Hotta et al.'s finding. We also determine the three percentages considering the 12 decision points under *CCFinderX-Combined* heading. These percentages are: 50%, 50% and 0% respectively for Category 1, 2, and 3. As the percentages regarding Category

TABLE V: Modification Frequencies by Hotta et al.'s methodology

	Systems	Type 1			Type 2			Type 3			NiCad-Combined			CCFinderX-Combined		
		MF_c	MF_n	Rem	MF_c	MF_n	Rem	MF_c	MF_n	Rem	MF_c	MF_n	Rem	MF_c	MF_n	Rem
Java	DNSJava	27.91	8.69	⊖	11.93	10.48	○	10.62	10.61	○	10.40	10.66	○	16.94	7.66	⊖
	Ant-Contrib	9.96	4.09	⊖	3.18	4.27	⊕	3.76	4.41	⊕	4.12	4.4	○	2.58	4.91	⊕
	Carol	10.11	18.05	⊕	11.70	11.70	○	19.52	17.21	⊖	26.06	15.55	⊖	21.89	16.32	⊖
	jabref	17.67	22.46	⊕	16.38	22.53	⊕	19.83	22.68	⊕	20.23	20.86	○	62.06	7.59	⊖
C	Ctags	11.13	7.83	⊖	13.48	7.78	⊖	9.37	7.81	⊖	10.16	7.74	⊖	8.932	7.86	⊖
	Camellia	18.50	18.04	○	42.37	17.73	⊖	30.02	17.53	⊖	34.59	17.31	⊖	47.02	14.20	⊖
	QMail Admin	45.99	51.50	⊕	56.41	50.75	⊖	56.14	50.69	⊖	50.95	51.36	○	39.15	22.07	⊖
	Gnumakeuniproc	32.25	39.74	⊕	74.06	37.70	⊖	78.64	36.50	⊖	35.04	38.82	⊕	35.69	40.33	⊕
C#	GreenShot	4.13	9.62	⊕	7.62	9.59	⊕	8.65	9.60	⊕	8.54	9.63	⊕	8.35	9.85	⊕
	ImgSeqScan	0	24.97	⊕	0	25.28	⊕	0	25.43	⊕	0	25.67	⊕	11.82	28.68	⊕
	Capital Resource	0	31.69	⊕	0	31.37	⊕	12.18	31.48	⊕	9.87	31.57	⊕	22.61	34.25	⊕
	MonoOSC	7.09	22.71	⊕	15.18	22.53	⊕	13.09	22.78	⊕	13.03	22.84	⊕	15.46	23.59	⊕

MF_c= Modification Frequency of Cloned Code *MF_n= Modification Frequency of Non-Cloned Code* *Rem = Remark*
⊕= *$MF_c < MF_n$ (Category 1, CLONES MORE STABLE)* ⊖= *$MF_c > MF_n$ (Category 2, CLONES LESS STABLE)*
○= *The decision point falls in Category 3*

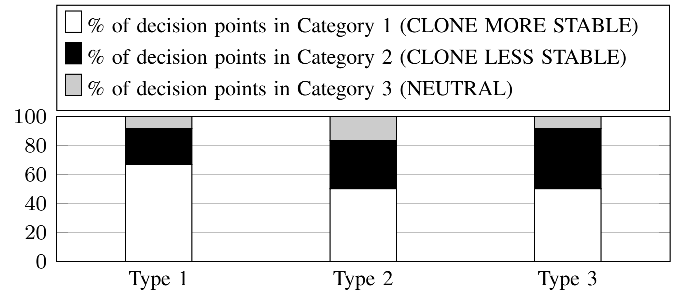


Fig. 4: Type centric statistics for modification frequency.

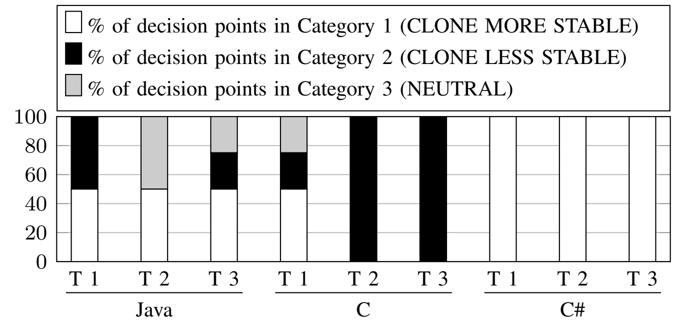


Fig. 5: Language-wise type centric statistics for modification frequency

1 and 2 are the same, we could not come to a concluding decision about the comparative modification frequencies of cloned and non-cloned code from CCFinderX results. We show the percentages for the two clone detectors in Fig. 2.

Language Centric Analysis: For language centric analysis on Table V, we present the graph in Fig. 3. We see that in case of both Java and C languages, the percentage of decision points in Category 2 is higher than the corresponding percentage in Category 1 for each clone detector. From Table V we realize that the overall percentages in Fig. 2 were mostly influenced by the Category 1 decision points from C# language. From Fig. 3 we see that all the decision points regarding C# are of Category 1 (CLONES MORE STABLE). From language centric analysis we finally decide that *code clones in both Java and C programming languages exhibit higher instability than non-cloned code. The opposite is true for C# language.*

Type Centric Analysis: From the type centric statistics in Fig. 4 constructed from Table V we see that for each of the three clone-types (Type 1, Type 2, and Type 3), the highest proportion of decision points belong to Category 1 (CLONES MORE STABLE) and agree with lower modification frequency of cloned code. Thus, we can say that *each of the three types of clones are likely to receive less frequent changes compared to the non-cloned code in general.*

Type Centric Analysis for Each Language: For this analysis we draw the graph in Fig. 5 from Table V. According to this graph, *both Type 2 and Type 3 clones of programming language C have a very high probability of getting more frequent changes compared to non-cloned code.* From the graphs in Fig. 4, and 5 we can say that *although each of the three types of clones (Type 1, Type 2, Type 3) appear to receive less frequent changes in general, Type 2 and Type 3 clones are likely to receive more frequent changes than non-cloned code in case of the subject systems written in C.*

Answer to RQ 1: According to our overall analysis involving the combined-type results, *code clones generally get modified less frequently compared to non-cloned code.* Such a finding complies with the finding in the original study done by Hotta et al. [24]. They used CCFinderX [28] clone detector. We have also used it. We have used another clone detector, the NiCad clone detector [16], which was not used in the original study [24]. NiCad can detect code clones by separating them by clone-types, and also, it can detect Type 3 clones. CCFinderX cannot detect Type 3 clones. As we use NiCad clone detector, we perform clone-type centric analysis of our experimental results. We also perform language centric analysis. Language centric and clone-type centric analyses were not performed in Hotta et al's study [24].

From our language centric analysis (Fig. 3) we find that *code clones in Java and C systems often get modified more frequently compared to non-cloned code in such systems. Type 2 and Type 3 clones in C systems should be given higher priority for management, because such*

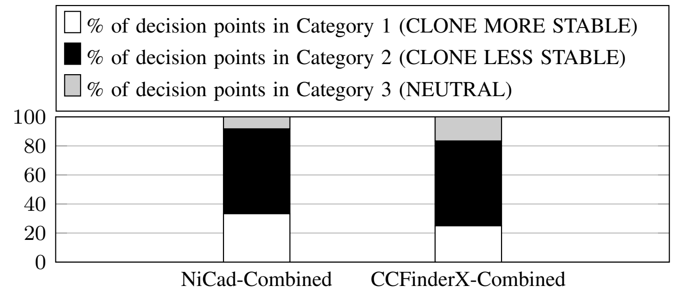


Fig. 6: Overall statistics for modification probability.

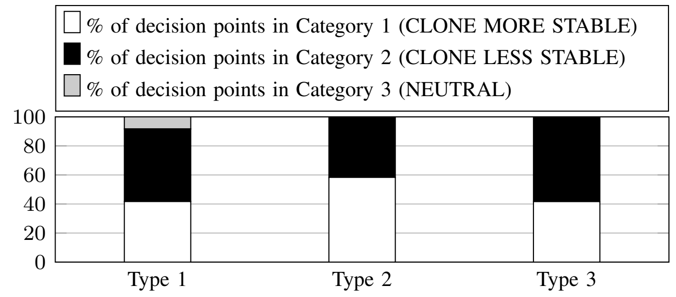


Fig. 8: Type centric statistics for modification probability.

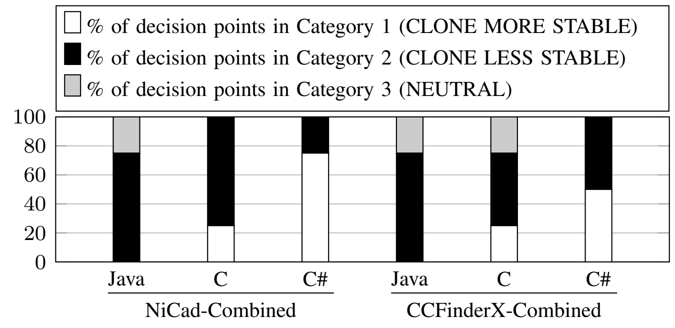


Fig. 7: Language centric statistics for modification probability

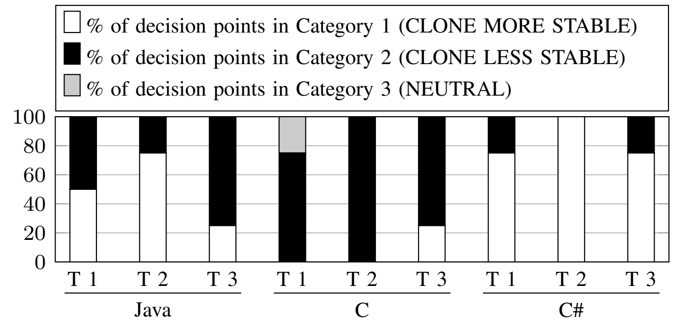


Fig. 9: Language-wise type centric statistics for modification probability

clones have always been more unstable (i.e., have always been modified more frequently) than non-cloned code according to our analysis (Fig. 5).

B. Analysis of the Experimental Results Regarding Modification Probability

Using our implementation of Göde et al.'s [19] methodology we calculated the modification probabilities of cloned and non-cloned code of our candidate systems using the equations: Eq. 3 and Eq. 4 and populated the results in Table VI. For a particular significant decision point in this table,

(i) if $MP_c < MP_n$, then clone has lower probability of getting changes compared to the probability of non-cloned code and this point falls in Category 1 (CLONES MORE STABLE)

(ii) if $MP_c > MP_n$, then clone has higher probability of getting changes compared to the probability of non-cloned code and this point falls in Category 2 (CLONES LESS STABLE)

We answer the second research question RQ 2 from four dimensions as follows.

Overall analysis: We show the overall statistics for modification probability in the graph of Fig. 6 considering the combined-type results in Table VI. We see that each of the two clone detectors (NiCad and CCFinderX) suggest *a higher modification probability of cloned code compared to non-cloned code*. In other words, *the proportion of source code lines affected in the clone regions per commit operation is generally greater than the proportion of lines affected in the non-clone regions.*

Language centric analysis: We draw the graph in Fig. 7 from Table VI for this analysis. We see that in the cases of Java and C languages the proportion of decision points belonging to Category 2 (CLONE LESS STABLE) is much higher than the proportion of decision points belonging to Category 1 (CLONE MORE STABLE). In the cases of Java, no decision points belong to Category 1. Thus, *code clones in both Java and C systems exhibit higher modification probability than non-cloned code in general.* From the bars regarding C# it seems that *code clones in C# systems have a tendency of showing a lower modification probability compared to non-cloned code.*

Type Centric Analysis: We constructed the graph in Fig. 8 from Table VI for observing the type centric statistics. According to this graph, *both Type 1 and Type 3 clones have a slightly higher modification probability compared to the Type 2 clones.*

Type centric analysis for each language: We draw the graph in Fig. 9 from Table VI for this analysis. According to the graph, *each of the three clone-types (Type 1, Type 2, and Type 3) of C exhibits higher modification probability compared to non-cloned code.* However, *in case of Java, only Type 3 is unstable* because, 75% of the points belonging to Type 3 shows a higher modification probability of cloned code. We also see that *code clones of each clone-type in C# are generally more stable than non-cloned code.*

TABLE VI: Modification Probability by Göde et al.'s methodology

Lang	Systems	Type 1			Type 2			Type 3			NiCad-Combined			CCFinderX-Combined		
		MP_c	MP_n	Rem	MP_c	MP_n	Rem	MP_c	MP_n	Rem	MP_c	MP_n	Rem	MP_c	MP_n	Rem
Java	DNSJava	0.004926	0.000641	⊖	0.002260	0.001636	⊖	0.001980	0.001628	⊖	0.001942	0.001626	⊖	0.002786	0.001116	⊖
	Ant-Contrib	0.005800	0.000728	⊖	0.000359	0.000855	⊕	0.001079	0.000796	⊖	0.001454	0.000723	⊖	0.000844	0.000827	○
	Carol	0.000599	0.000818	⊕	0.000521	0.000826	⊕	0.000928	0.000780	⊖	0.001132	0.000682	⊖	0.001050	0.000716	⊖
	jabref	0.000390	0.000466	⊕	0.000412	0.000467	⊕	0.000405	0.000469	⊕	0.000256	0.000246	○	0.001555	0.000099	⊖
C	Ctags	0.001048	0.000958	⊖	0.001157	0.000955	⊖	0.000689	0.000981	⊕	0.000790	0.000974	⊕	0.000943	0.000962	○
	Camellia	0.001655	0.000572	⊖	0.002405	0.000567	⊖	0.001539	0.000551	⊖	0.001665	0.000536	⊖	0.001712	0.000425	⊖
	QMail Admin	0.005822	0.003326	⊖	0.006150	0.003326	⊖	0.007129	0.003052	⊖	0.006266	0.00304	⊖	0.002419	0.001444	⊖
	Gnumake Uniproc	0.000304	0.000328	○	0.000590	0.000319	⊖	0.000660	0.000308	⊖	0.000367	0.000319	⊖	0.000309	0.000543	⊕
C#	GreenShot	0.001360	0.000934	⊖	0.000778	0.000944	⊕	0.001000	0.000938	⊖	0.001130	0.000929	⊖	0.001087	0.000920	⊖
	ImgSeqScan	0	0.024880	⊕	0	0.025192	⊕	0	0.025340	⊕	0	0.0255783	⊕	0.014523	0.027462	⊕
	Capital Resource	0	0.000988	⊕	0	0.000978	⊕	0.000209	0.000983	⊕	0.000169	0.000986	⊕	0.001175	0.000912	⊖
	MonoOSC	0.001018	0.003862	⊕	0.001246	0.003833	⊕	0.001177	0.003902	⊕	0.001289	0.003914	⊕	0.001618	0.004126	⊕

MP_c= Modification Probability of Cloned Code *MP_n= Modification Probability of Non-Cloned Code* *Rem = Remark*
⊕= *$MP_c < MP_n$ (Category 1, CLONES MORE STABLE)* ⊖= *$MP_c > MP_n$ (Category 2, CLONES LESS STABLE)*
○= *The decision point falls in Category 3*

Answer to RQ 2: Our analysis on the combined-type clone detection results (Fig. 6) indicates that *code clones generally have a higher probability of getting modified than non-cloned code*. Such a finding contradicts with the finding of the original study done by Göde et al. [19]. Göde et al. used iClones [20] in their study, and investigated only two systems written in Java. We have used CCFinderX [28] and NiCad [16] in our study, and investigate 12 systems covering three programming languages: Java, C, and C#. Possibly, these are the reasons why our finding contradicts with the original one. We believe that our finding is more generalized with two clone detectors and a wide variety of subject systems.

Göde et al. did not perform any clone-type centric or language centric analysis. We perform such analyses in our study. We find (Fig. 7) that *code clones in Java and C systems exhibit higher possibility of getting modified compared to non-cloned code*. From Fig. 9 we see that *Type 3 clones of Java systems, and all three types of clones in C systems exhibit higher modification probability than non-cloned code. Code clones in C# systems appear to be more stable than non-cloned code in such systems.*

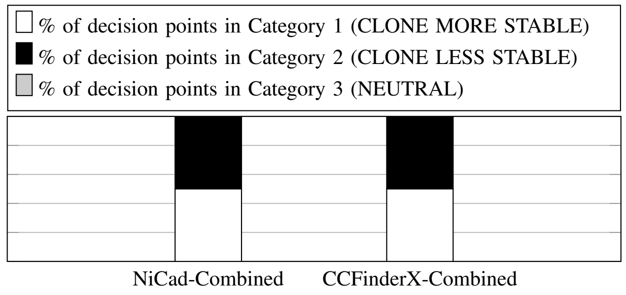


Fig. 10: Overall analysis for average last change date.

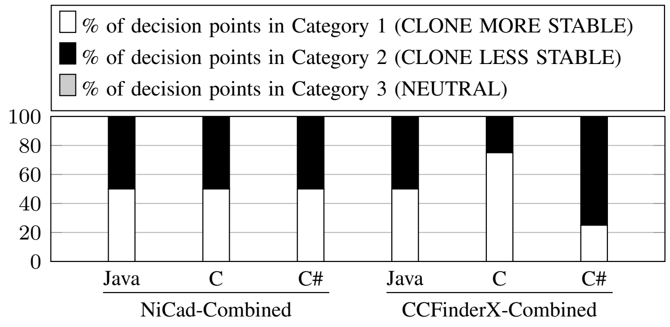


Fig. 11: Language centric statistics for average last change date

C. Analysis of the Experimental Results for Average Last Change Date

We calculated the average last change dates of the cloned ($ALCD_c$) and non-cloned ($ALCD_n$) code of the selected last revisions (Table IV) of the candidate subject systems by applying our implementation of Krinke's Methodology [33]. $ALCD_c$ and $ALCD_n$ were calculated using the equations Eq. 5 and Eq. 6 respectively. Table VII contains both combined type and individual type results. We mentioned (with explanation) that for categorizing the decision points in the tables for this metric we cannot use the eligibility value calculation technique [42]. If two dates corresponding to a particular decision point are different, we considered the decision point as a significant one. For a particular significant decision point in Table VII,

(i) if $ALCD_c$ is older than $ALCD_n$, then this point falls in Category 1 (CLONES MORE STABLE), because for this point, changes to non-cloned code are more recent on an average compared to the changes to the cloned code.

(ii) if $ALCD_n$ is older than $ALCD_c$, then this point falls in Category 2 (CLONES LESS STABLE), because for this point, changes to the cloned code are more recent on an average compared to changes to non-cloned code.

The following four sections of analysis answer the third research question RQ 3 from four dimensions.

Overall analysis: We draw the graph in Fig. 10 for

TABLE VII: Average Last Change Dates by Krinke's methodology

Lang	Systems	Type 1			Type 2			Type 3			NiCad-Combined			CCFinderX-Combined		
		$ALCD_c$	$ALCD_n$	R	$ALCD_c$	$ALCD_n$	R	$ALCD_c$	$ALCD_n$	R	$ALCD_c$	$ALCD_n$	R	$ALCD_c$	$ALCD_n$	R
Java	DNSJava	26-Aug-04	4-Jul-04	⊖	16-Nov-04	2-Jul-04	⊖	8-Dec-04	1-Jul-04	⊖	18-Nov-04	1-Jul-04	⊖	17-Aug-04	2-Jul-04	⊖
	Ant-Contrib	14-Sep-06	11-Aug-06	⊖	17-Jul-06	11-Aug-06	⊕	1-Sep-06	10-Aug-06	⊖	26-Aug-06	10-Aug-06	⊖	16-Dec-06	1-Aug-06	⊖
	Carol	8-Sep-07	22-Aug-07	⊖	7-Sep-07	22-Aug-07	⊖	6-Aug-07	23-Aug-07	⊕	14-Aug-07	23-Aug-07	⊕	14-Aug-07	23-Aug-07	⊕
	jabref	13-Jun-06	4-Jun-06	⊖	7-Feb-06	5-Jun-06	⊕	13-May-06	5-Jun-06	⊕	03-May-06	05-Jun-06	⊕	21-Feb-06	8-Jun-06	⊕
C	Ctags	18-Mar-08	30-Dec-06	⊖	29-Dec-06	31-Dec-06	⊕	05-Jan-07	31-Dec-06	⊖	24-Oct-06	27-Mar-07	⊕	15-Nov-05	04-Jan-07	⊕
	Camellia	4-Nov-07	14-Nov-07	⊕	17-Jul-08	14-Nov-07	⊖	8-Feb-09	9-Nov-07	⊖	5-Oct-08	10-Nov-07	⊖	26-Jan-08	13-Nov-07	⊖
	QMail Admin	7-Nov-03	27-Oct-03	⊖	13-Nov-03	27-Oct-03	⊖	11-Nov-03	27-Oct-03	⊖	2-Dec-03	27-Oct-03	⊖	13-Oct-03	28-Oct-03	⊕
	Gnumake Uniproc	27-Aug-09	28-Sep-09	⊕	18-Oct-09	27-Sep-09	⊖	19-Oct-09	27-Sep-09	⊖	10-Jul-09	18-Oct-09	⊕	22-Jul-09	04-Oct-09	⊕
C#	GreenShot	8-Jun-10	18-Jun-10	⊕	19-Jun-10	18-Jun-10	⊖	23-Jun-10	18-Jun-10	⊖	23-Jun-10	18-Jun-10	⊖	13-Jun-10	19-Jun-10	⊕
	ImgSeqScan	19-Jan-11	13-Jan-11	⊖	14-Jan-11	13-Jan-11	⊖	19-Jan-11	13-Jan-11	⊖	19-Jan-11	13-Jan-11	⊖	18-Jan-11	12-Jan-11	⊖
	Capital Resource	13-Dec-08	12-Dec-08	⊖	10-Dec-08	12-Dec-08	⊕	11-Dec-08	12-Dec-08	⊕	11-Dec-08	12-Dec-08	⊕	13-Dec-08	12-Dec-08	⊖
	MonoOSC	25-Jun-09	21-Mar-09	⊖	14-Mar-09	21-Mar-09	⊕	26-Dec-08	22-Mar-09	⊕	14-Mar-09	31-Mar-09	⊕	22-Apr-09	01-Mar-09	⊖

$ALCD_c$= Average Last Change Date of Cloned Code *$ALCD_n$= Average Last Change Date of Non-Cloned Code* *R = Remark*
⊕= $ALCD_c$ is older than $ALCD_n$ (Category 1, CLONES MORE STABLE)
⊖= $ALCD_c$ is newer than $ALCD_n$ (Category 2, CLONES LESS STABLE) *○= The decision point falls in Category 3*

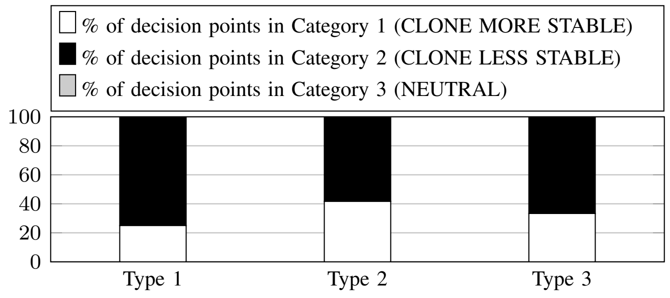


Fig. 12: Type centric statistics for average last change date.

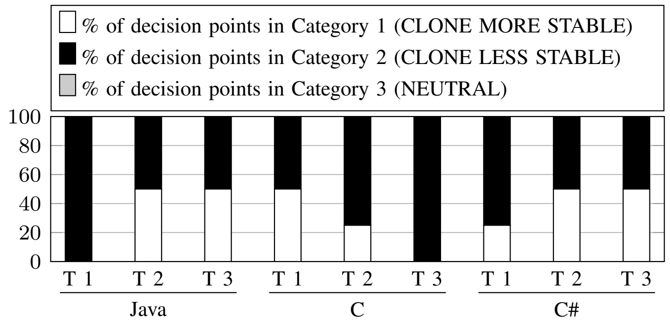


Fig. 13: Language-wise type centric statistics for average last change date

over analysis. We see that in case of each clone detectors, 50% of the decision points belonging to Category 1. The remaining 50% of the points belong to Category 2. The figure indicates that *cloned code can sometimes be changed more lately compared to non-cloned code during the evolution of a software system.*

Language centric analysis: Fig. 11 constructed from Table VII helps us make a language centric decision for this metric. According to the clone detection results of CCFinderX, *cloned code in the subject systems written in C# has a higher probability of being changed more lately compared to non-cloned code. In other words, the clones in C# appear to be more unstable compared to non-cloned code. An opposite scenario has been shown by the subject systems written in C. In case of Java, the same proportion of decision points belong to both of the two categories: Category 1 and Category 2.* According to NiCad-combined type results, the percentages of decision points belonging to Category 1 and 2 are the same for each language.

Type centric analysis: According to the type centric statistics of the graph in Fig. 12, *each of the three types of clones are likely to be modified more lately compared to non-cloned code*. The graph also indicates that *both Type 1 and Type 3 clones have higher probability of getting more recent changes in comparison with the Type 2 clones.*

Type Centric Analysis for Each Language: According to the graph in Fig. 13, for all of the significant points belonging to Type 1 case of Java, average last change date of cloned code is younger than that of non-cloned code. The same is true for Type 3 clones of C. Type 2 clones of C and Type 1 clones of C# also show higher probabilities of being changed more lately. For all other cases, both cloned and non-cloned code have equal possibilities of being more lately changed. So, according to this metric, *each of the three clone-types of the selected programming languages is a threat to software stability during the maintenance phase with Type 1 clones of Java and Type 3 clones of C being the most unstable ones.*

Answer to RQ 3: Our overall analysis involving the combined-type results implies that *cloned and non-cloned code have equal probabilities of getting changed.* In the original study Krinke [33] found that non-cloned code gets modified more lately than cloned code. However, this study was done using Simian clone detector [59] which can detect Type 1 clones only. We use CCFinderX and NiCad clone detectors in our study. While CCFinderX provides combined-type results including Type 1 and Type 2 clones, NiCad detects three types of clones (Type 1, Type 2, Type 3) and reports them by combining or separating them by clone-types. The original study [33] was done on only three Java systems. We investigate 12

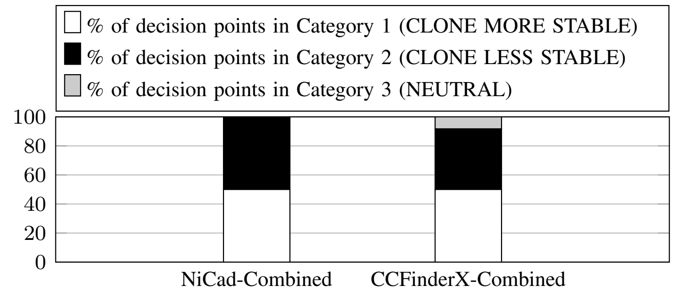
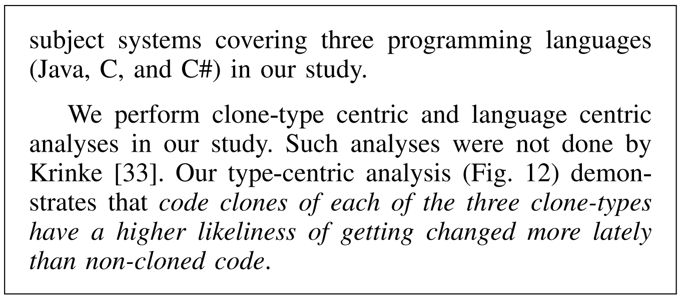


Fig. 14: Overall analysis for average age.



subject systems covering three programming languages (Java, C, and C#) in our study.

We perform clone-type centric and language centric analyses in our study. Such analyses were not done by Krinke [33]. Our type-centric analysis (Fig. 12) demonstrates that *code clones of each of the three clone-types have a higher likeliness of getting changed more lately than non-cloned code.*

D. Analysis of the Experimental Results Regarding Average Age

By applying the variant [41] of Krinke's methodology [33] we calculated the arithmetic average ages (in days) of cloned (AA_c) and non-cloned (AA_n) code of a subject system considering its last revision as mentioned in the Table IV. AA_c and AA_n were calculated according to the equations: Eq. 8 and Eq. 9 respectively. We present our obtained data for this metric in Table VIII. We did not apply the eligibility value calculation technique proposed by Mondal et al. [39] for this metric. In case of the other metrics where we applied the technique, the metric values represent the change-proneness of the code base. However, in this particular metric (Average Age), the metric values are ages in days. For a particular decision point, if the average ages of cloned and non-cloned code are different by at least one day we considered the decision point as a significant one. For a particular significant decision point in Table VIII,

(i) if $AA_c > AA_n$ by at least one day, then cloned code is more stable than non-cloned code for this point and this point belongs to Category 1 (CLONES MORE STABLE).

(ii) if $AA_c < AA_n$ by at least one day, then cloned code is less stable than non-cloned code for this point and this point belongs to Category 2 (CLONES LESS STABLE).

(iii) if the difference between AA_c and AA_n is less than one day, then this point belongs to Category 3 (NEUTRAL).

We answer the fourth research question RQ-4 in the following four sections.

Overall analysis: We draw the graph in Fig. 14 for our overall analysis. We see that 50% of the decision points in each case (NiCad-Combined and CCFinderX-Combined) suggest that cloned code remains unchanged for longer time compared to non-cloned code. These points fall in Category 1 (CLONES MORE STABLE). However, from the graph we realize that *code clones can sometimes exhibit higher instability than non-cloned code during evolution.*

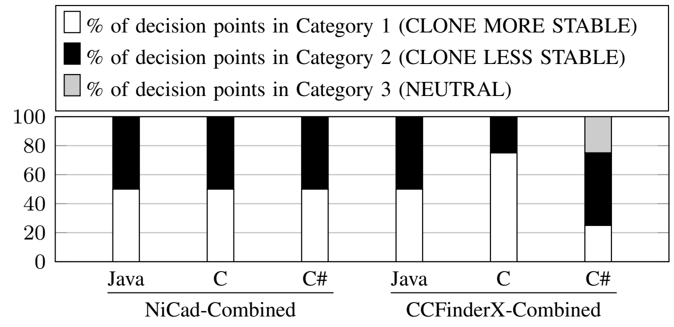


Fig. 15: Language centric statistics for average age

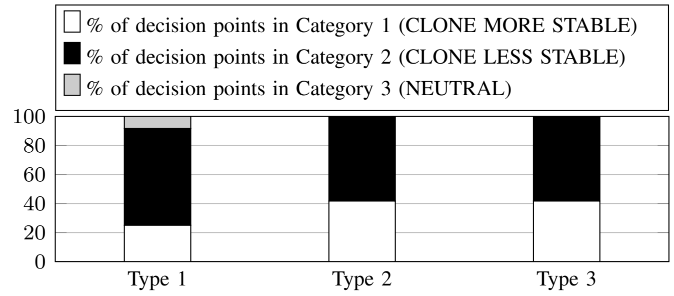


Fig. 16: Type centric statistics for average age.

Language centric analysis: The language centric statistics (Fig. 15 constructed from Table VIII) of this metric are almost the same as those of average last change date. The reason behind this is that each of these metrics are computed considering the last revision of a candidate subject system. We see that *code clones in C# appear to be more unstable compared to non-cloned code according to CCFinderX results. An opposite scenario is exhibited by the decision points in C (CCFinderX results). In case of Java, the same proportion of decision points (50%) belong to both Category 1 (CLONES MORE STABLE) and Category 2 (CLONES LESS STABLE).* According to NiCad results, 50% of the decision points belong to both of the categories: Category 1 and Category 2 for each language.

Type centric analysis: The type centric statistics (Fig. 16)

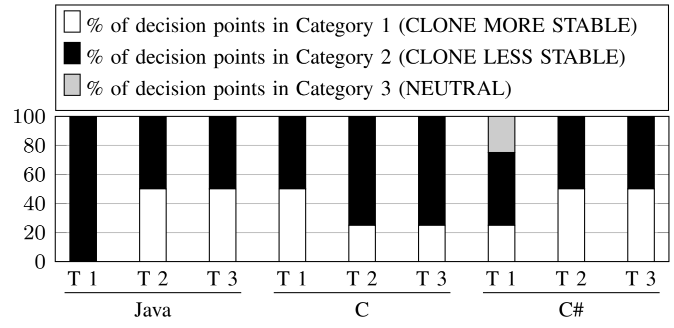


Fig. 17: Language-wise type centric statistics for average age

TABLE VIII: Average Ages of Cloned and Non-cloned Code

Lang	Systems	Type 1			Type 2			Type 3			NiCad-Combined			CCFinderX-Combined		
		AA_c	AA_n	Rem	AA_c	AA_n	Rem	AA_c	AA_n	Rem	AA_c	AA_n	Rem	AA_c	AA_n	Rem
Java	DNSJava	2386.51	2440.31	⊖	2304.85	2442	⊖	2282.94	2443.23	⊖	2303.49	2443.15	⊖	2395.84	2442.42	⊖
	Ant-Contrib	869.21	903.39	⊖	927.56	902.93	⊕	881.61	903.73	⊖	887.52	903.65	⊖	776.27	912.83	⊖
	Carol	193.61	210.75	⊖	194.65	210.85	⊖	226.65	210.04	⊕	218.55	210.24	⊕	218.89	209.82	⊕
	jabref	1063.87	1072.57	⊖	1189.77	1071.64	⊕	1094.87	1072.13	⊕	1104.97	1071.88	⊕	1175.59	1068.58	⊕
C	Ctags	1050.05	1345.77	⊖	1425.36	1344.75	⊕	1433.15	1344.46	⊕	1498.10	1344.02	⊕	1798.81	1339.45	⊕
	Camellia	1066.84	1056.77	⊕	810.96	1057.39	⊖	604.95	1062.48	⊖	730.55	1060.50	⊖	984.28	1057.77	⊖
	QMail Admin	2664.22	2674.61	⊖	2658.24	2674.62	⊖	2660.30	2674.63	⊖	2638.87	2674.94	⊖	2689.10	2674.36	⊕
	Gnumakeuniproc	255.37	221.64	⊕	213.91	222.74	⊖	212.97	222.74	⊖	301.02	200.93	⊕	288.06	215.16	⊕
C#	GreenShot	292.88	282.61	⊕	282.24	283.24	⊖	278.35	283.30	⊖	278.35	283.30	⊖	287.56	281.98	⊕
	ImgSeqScan	14.0	20.26	⊖	18.66	20.25	⊖	14.39	20.37	⊖	14.39	20.37	⊖	14.55	20.65	⊖
	Capital Resource	86.35	86.59	○	89.09	86.57	⊕	88.03	86.56	⊕	88.03	86.56	⊕	85.65	86.64	○
	MonoOSC	224.43	314.06	⊖	325.65	313.54	⊕	389.37	312.49	⊕	330.32	313.41	⊕	276.45	341.93	⊖

AA_c= *Average Age of Cloned Code.* AA_n= *Average Age of Non-cloned Code.* *Rem = Remark*
⊕= $AA_c > AA_n$ *(Category 1, CLONES MORE STABLE)* ⊖= $AA_c < AA_n$ *(Category 2, CLONES LESS STABLE)*
○= *The decision point falls in Category 3*

regarding *average age* follows the type centric statistics of *average last change date*. The reason behind this has already been described in the language centric analysis regarding average age (AA). We see that *each of the three types of clones has higher probability of getting changed more lately compared to non-cloned code.* In other words, *each clone-type appears to be more unstable than non-cloned code according to this metric*.

Type Centric Analysis for Each Language: The language-wise type centric analysis result is shown in the graph of Fig. 17 constructed from Table VIII. This graph is almost the same as that we obtained for average last change date. All of the decision points belonging to Type 1 case of Java indicate smaller longevity of cloned code. For Type 2 and Type 3 cases of C and Type 1 case of C#, major portions of the decision points suggest that non-cloned code lives longer than cloned code. For the remaining 5 cases, both cloned and non-cloned code exhibit equal probability of being unstable. So, according to this metric, *each of the clone types of each candidate programming language exhibits high probability of being more unstable compared to non-cloned code*.

We see that the language centric statistics (Fig. 15) is not agreeing with both type centric statistics (Fig. 16) and language-wise type centric statistics (Fig. 17). While language centric statistics suggest cloned code to be more stable than non-cloned code for two languages Java and C, type centric statistics for these two languages (Fig. 17) suggest cloned code to be less stable in general. The fact is that Fig. 15 and Fig. 17 have been constructed from two different sets of decision points (Fig. 15 is constructed considering the decision points under the headings **NiCad-Combined** and **CCFinderX-Combined**. Fig. 17 is constructed considering the points under the headings **Type 1**, **Type 2** and **Type 3**.) in Table VIII. While Fig. 15 reflects the combined type results of both NiCad and CCFinderX, Fig. 17 reflects the individual type results of NiCad only.

Answer to RQ 4: Our overall analysis on the combined-type results suggests that *code clones often have smaller longevity than non-cloned code during evolution*. Combined-type results were not investigated in the original study [41]. Only the individual-type results of NiCad clone detector [16] were considered. Our type-centric analysis suggests that *each of the three types of clones exhibits higher instability than non-cloned code in general*. Such a finding complies with the finding in the original study [41].

E. Analysis of the Experimental Results Regarding Impact and Likelihood

We calculate the followings using our implementation of the methodology proposed by Lozano and Wermelinger [37].

(1) Impact of cloned code (ICC) using Eq. 10.

(2) Impact of non-cloned code (INC) using Eq. 12.

(3) Likelihood of cloned code (LCC) using Eq. 14.

(4) Likelihood of non-cloned code (LNC) using Eq. 16.

Here, we should note that Lozano and Wermelinger discarded 2.5% of the largest commit operations from each of their subject systems in order to eliminate the effects of atypical changes [37]. An atypical change is a change that affects multiple functionalities. We have also discarded 2.5% of the largest commits from each of our candidate systems when calculating the impact and likelihood measures. These measures for cloned and non-cloned code are shown in Tables IX, X respectively. For a particular significant decision point contained in Table IX,

(i) if $ICC < INC$, then this point falls in Category 1 (CLONES MORE STABLE)

(ii) if $ICC > INC$, then this point falls in Category 2 (CLONES LESS STABLE)

Also, for a particular significant decision point contained in Table X,

(i) if $LCC < LNC$, then this point falls in Category 1, because for this point, cloned code is less likely to be changed compared to non-cloned code.

(ii) if $LCC > LNC$, then this point falls in Category 2, because for this point, cloned code is more likely to be changed than non-cloned code.

TABLE IX: Impact of cloned and non-cloned code by the methodology of Lozano and Wermelinger

Lang	Systems	Type 1			Type 2			Type 3			NiCad-Combined			CCFinderX-Combined		
		ICC	INC	Rem	ICC	INC	Rem	ICC	INC	Rem	ICC	INC	Rem	ICC	INC	Rem
Java	DNSJava	0.004306	0.001593	⊖	0.003644	0.002289	⊖	0.003378	0.002441	⊖	0.003330	0.002450	⊖	0.003093	0.002309	⊖
	Ant-Contrib	0.035714	0.014431	⊖	0.017205	0.014818	⊖	0.016187	0.015604	○	0.015564	0.015398	○	0.014623	0.014612	○
	Carol	0.002214	0.001302	⊖	0.002166	0.001318	⊖	0.002670	0.001446	⊖	0.002528	0.001489	⊖	0.002269	0.001651	⊖
	jabref	0.003630	0.003038	⊖	0.003237	0.003097	○	0.003648	0.003116	⊖	0.004282	0.003048	⊖	0.003453	0.002901	⊖
C	Ctags	0.003461	0.002582	⊖	0.002533	0.002818	⊕	0.002510	0.002524	○	0.002770	0.002547	○	0.002832	0.002869	○
	Camellia	0.014529	0.013044	⊖	0.013855	0.013020	○	0.014032	0.013155	○	0.013323	0.013170	○	0.013722	0.012873	○
	QMail Admin	0.037046	0.027128	⊖	0.120279	0.013481	⊖	0.031811	0.020772	⊖	0.030669	0.040555	⊕	0.011889	0.042659	⊕
	Gnumake Uniproc	0.030799	0.027795	⊖	0.025762	0.025433	○	0.061090	0.025809	⊖	0.062269	0.026350	⊖	0.026147	0.023723	⊖
C#	GreenShot	0.003980	0.003020	⊖	0.003310	0.003074	○	0.003917	0.003133	⊖	0.003607	0.003220	⊖	0.003030	0.002906	○
	ImgSeqScan	0	0.056784	⊕	0	0.051490	⊕	0	0.051490	⊕	0	0.051490	⊕	0.035525	0.067790	⊕
	Capital Resource	0	0.011515	⊕	0	0.011515	⊕	0.012300	0.011517	○	0.012300	0.011517	○	0.011333	0.011554	○
	MonoOSC	0.004739	0.007185	⊕	0.055743	0.007391	⊖	0.005750	0.007577	⊕	0.005366	0.007618	⊕	0.009784	0.007366	⊖

ICC= Impact of Cloned Code *INC= Impact of Non-Cloned Code* *Rem = Remark*
⊕= *ICC <INC (Category 1, CLONES MORE STABLE)* ⊖= *ICC >INC (Category 2, CLONES LESS STABLE)*
○= *The decision point falls in Category 3*

TABLE X: Likelihood of changes of cloned and non-cloned methods by the methodology of Lozano and Wermelinger

Lang	Systems	Type 1			Type 2			Type 3			NiCad-Combined			CCFinderX-Combined		
		LCC	LNC	Rem	LCC	LNC	Rem	LCC	LNC	Rem	LCC	LNC	Rem	LCC	LNC	Rem
Java	DNSJava	0.010465	0.006079	⊖	0.010300	0.004838	⊖	0.011197	0.006453	⊖	0.010594	0.006741	⊖	0.011812	0.005109	⊖
	Ant-Contrib	0.75	0.050016	⊖	0.053453	0.080826	⊕	0.168700	0.098941	⊖	0.152325	0.105432	⊖	0.064301	0.091879	⊕
	Carol	0.024513	0.007073	⊖	0.025205	0.007664	⊖	0.023643	0.009888	⊖	0.020661	0.010073	⊖	0.023418	0.011425	⊖
	jabref	0.012831	0.001861	⊖	0.014853	0.002014	⊖	0.009401	0.003026	⊖	0.008857	0.002179	⊖	0.004541	0.002599	⊖
C	Ctags	0.065153	0.008283	⊖	0.024657	0.008239	⊖	0.069739	0.013531	⊖	0.033359	0.013643	⊖	0.010325	0.007249	⊖
	Camellia	0.023832	0.020232	○	0.040466	0.019938	⊖	0.039178	0.019385	⊖	0.046392	0.019575	⊖	0.020016	0.035980	⊕
	QMail Admin	0.030872	0.022933	⊖	0.047893	0.018799	⊖	0.041239	0.021687	⊖	0.038644	0.016072	⊖	0.028739	0.018757	⊖
	Gnumake Uniproc	0.256817	0.140758	⊖	0.040816	0.081776	⊕	0.094228	0.068551	⊖	0.100809	0.080028	⊖	0.073717	0.099099	⊕
C#	GreenShot	0.125183	0.014503	⊖	0.022490	0.016716	⊖	0.035680	0.016386	⊖	0.022600	0.017880	⊖	0.061601	0.010974	⊖
	ImgSeqScan	0	0.304426	⊕	0	0.279315	⊕	0	0.279315	⊕	0	0.279315	⊕	0.531635	0.231955	⊖
	Capital Resource	0	0.032684	⊕	0	0.032684	⊕	0.012811	0.035023	⊕	0.012811	0.035023	⊕	0.058653	0.029643	⊖
	MonoOSC	0.061955	0.032516	⊖	0.119999	0.027509	⊖	0.018143	0.030559	⊕	0.032967	0.030469	○	0.074932	0.030858	⊖

LCC= Likelihood of Cloned Code *LNC= Likelihood of Non-Cloned Code* *Rem = Remark*
⊕= *LCC <LNC (Category 1, CLONES MORE STABLE)* ⊖= *LCC >LNC (Category 2, CLONES LESS STABLE)*
○= *The decision point falls in Category 3*

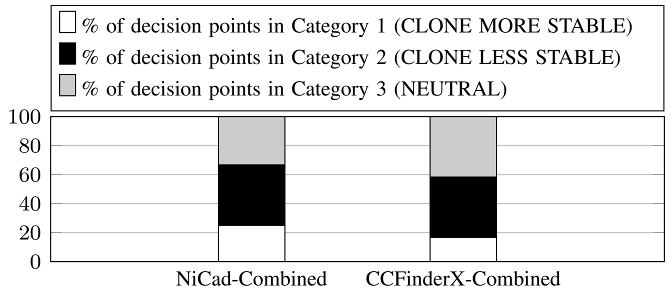


Fig. 18: Overall analysis for impact.

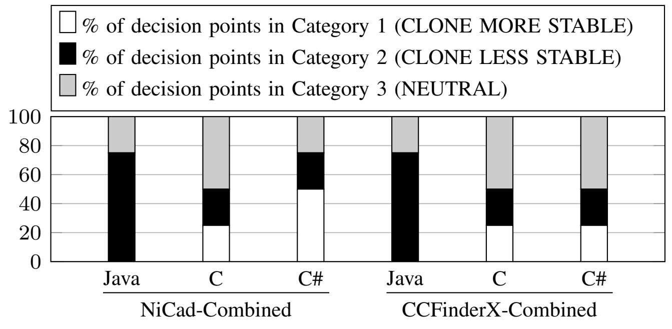


Fig. 19: Language centric statistics for impact

We answer the fifth and sixth research questions (RQ 5 and RQ 6) regarding impact and likelihood in the following eight sections.

Overall analysis for impact: We draw the graph in Fig. 18 for our overall analysis. We see that the percentage of decision points in Category 2 (CLONE LESS STABLE) is much higher than that in Category 1 (CLONE MORE STABLE) in the case of each clone detector. Such a scenario suggest that *cloned code has higher impact than non-cloned code in the maintenance phase*. In other words, *the average number of co-changed methods for a change in a cloned method is higher than the average number of co-changed methods for a change in a non-cloned method.*

Language centric analysis for impact: The language centric analysis for this metric is shown in the graph of Fig. 19. We see that all of the significant points belonging to Java programming language for each clone detector suggest that

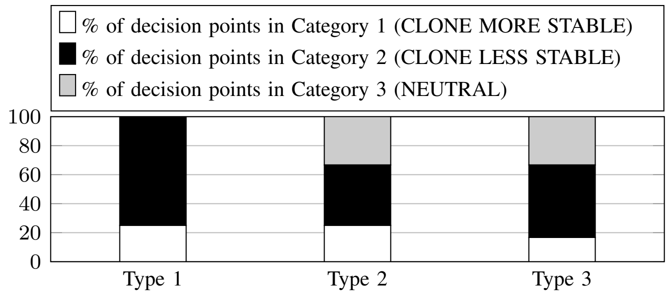


Fig. 20: Type centric statistics for impact.

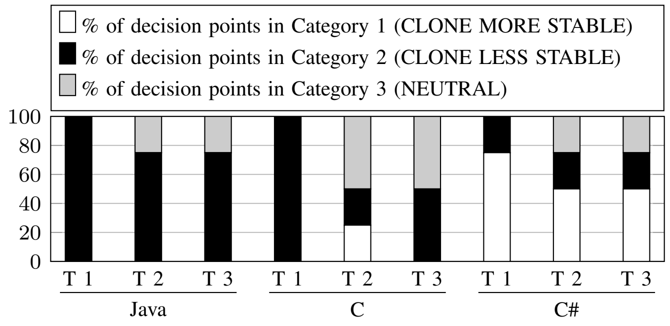


Fig. 21: Language-wise type centric statistics for impact

cloned code has higher impact than non-cloned code (Category 2, CLONES LESS STABLE). However, in case of C#, higher proportion of significant decision points suggest lower impact of cloned code (Category 1, CLONES MORE STABLE) according to NiCad results. For C, the same proportion (25%) of decision points belong to both Category 1 and Category 2 for each clone detector. According to this metric, *clones in Java programming language show higher instability compared to non-cloned code in the maintenance phase*.

Type centric analysis for impact: According to the type centric analysis regarding impact (Fig. 20), *for each of the three clone types, the impact of changing cloned methods is generally higher than the impact of changing non-cloned methods.* In other words, *according to this metric (impact), each type of clones is more unstable compared to the non-cloned code*.

Type Centric Analysis for Each Language Regarding Impact: According to the type centric statistics shown in the graph of Fig. 21, *all of the three clone types of Java are suggested to be highly unstable in the maintenance phase. The same is true for the Type 1 and Type 3 clones of C. In case of C#, each of the three types of clones exhibits lower impact than non-cloned code during maintenance phase.*

Answer to RQ 5: According to our analysis on the combined-type clone results, *the impact of changing a cloned method (i.e., the number of other methods that get changed as a result of changing a cloned method) is generally higher compared to the impact of changing a non-cloned method.* The original study [37] was performed using CCFinderX clone detection results (i.e., the combined type clone results of CCFinderX. CCFinderX cannot separate the clone detection results by clone-types). Our finding from overall analysis complies with the finding of the original study. We also performed language centric and type centric analysis. These analyses were not done in the original study.

According to our clone-type centric analysis, *cloned methods containing each of the three types of clones have a higher impact than non-cloned methods*. Our language centric analysis in Fig. 21 demonstrates that *cloned methods in both Java and C exhibit higher impact than non-cloned methods*.

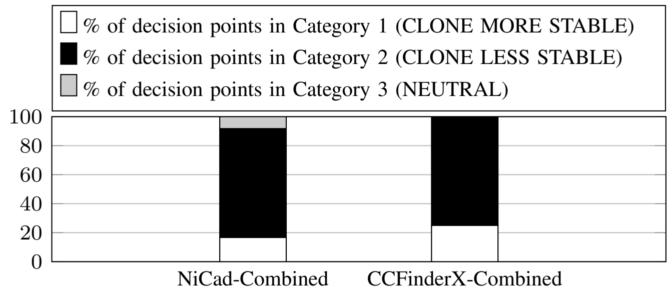


Fig. 22: Overall analysis for likelihood.

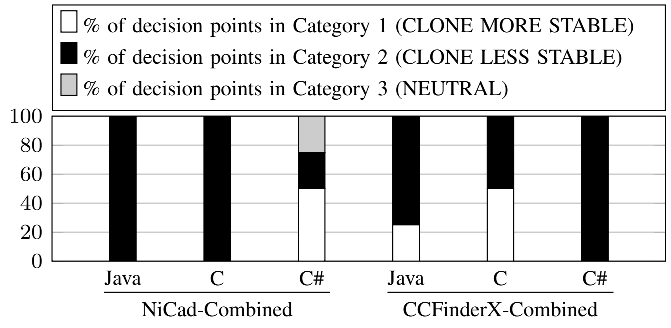


Fig. 23: Language centric statistics for likelihood

Overall analysis for likelihood: We draw the graph in Fig. 22 for our overall analysis. We see that the percentage of decision points in Category 2 (CLONE LESS STABLE) is much higher than that of Category 1 (CLONE MORE

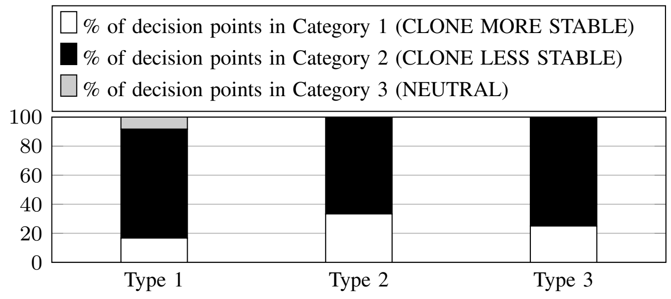


Fig. 24: Type centric statistics for likelihood.

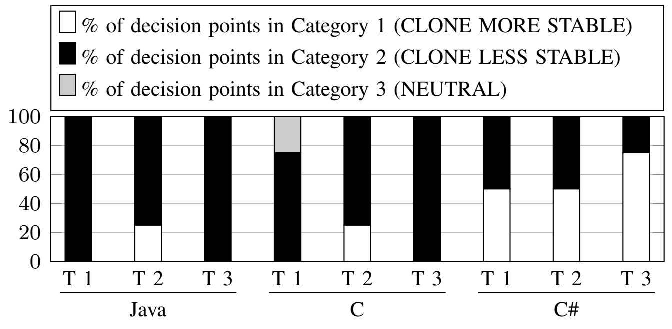


Fig. 25: Language-wise type centric statistics for likelihood

STABLE) for each clone detector. Such a scenario suggests that *cloned code is more likely to be changed than the non-cloned code in the maintenance phase*.

Language centric analysis for likelihood: According to the language centric statistics (Fig. 23) for both NiCad and CCFinderX results, code clones in Java systems have a very high likelihood of getting changed during evolution compared to non-cloned code. Code clones in C systems also have a high tendency of being more unstable than non-cloned code. In case of C#, while CCFinderX suggests a very high tendency of change-proneness of cloned code, NiCad suggests the opposite. We have mentioned that the combined-type clone results of NiCad are different than those of CCFinderX. NiCad results contain Type 3 clones. However, CCFinderX cannot detect Type 3 clones. Possibly, this difference is the primary reason of the different scenarios in case of C#.

Type centric analysis for likelihood According to the type centric statistics exhibited by the graph in Fig. 24, *for each of the three clone types, cloned code is more likely to get modified compared to non-cloned code*.

Type Centric Analysis for Each Language Regarding Likelihood: The language-wise type centric statistics of the graph in Fig. 25 strongly suggests that *each of the three clone-types of Java and C are highly unstable in the maintenance phase because they exhibit much higher likelihood of changes compared to non-cloned code. Also, 50% of the points belonging to both Type 1 and Type 2 case of C# suggest cloned code to be more unstable. However, Type 3 clones of this language (C#) are more stable than non-cloned code.*

Answer to RQ 6: According to our overall analysis, *cloned methods are generally more likely to change than non-cloned methods*. Such a finding complies with the finding of the original study performed by Lozano and Wermelinger [37].

The original study [37] does not contain any clone-type centric or language centric analysis of clone stability. We perform such analysis in our study. According to our analysis, *cloned methods containing any of the three types of clones in Java and C systems, exhibit a higher change-proneness than non-cloned methods*. According to Fig. 25, *cloned methods in C# systems seem to be less change-prone than non-cloned methods in these systems*.

F. Analysis of the Experimental Results Regarding Average Instability per Cloned Method

We calculated $EPCM$ and $CPCM$ according to the equations Eq. 18 and 19 using our implementation of Lozano and Wermelinger's study [36]. However, the values of $EPCM$ and $CPCM$ that we get using these equations are percentages. We normalize these values within zero to one by dividing them by 100.

These two metrics ($EPCM$ and $CPCM$) together can help us to take decision about the instabilities of cloned methods due to cloned and non-cloned code (Cloned methods might also contain non-cloned portions). The combined type and individual type results (normalized between zero to one) for these two metrics are presented in Table XI respectively. Our decision making procedure using these two metrics is explained below.

We have already mentioned that $EPCM$ is the average proportion of cloning in the cloned methods. Also, $CPCM$ is the average proportion of changes to the clones in the cloned methods. We take stability decisions comparing these two proportions in the following way.

(1) For a particular decision point, if the difference between $EPCM$ and $CPCM$ is not significant (the eligibility value calculated by Eq. 23 is less than the threshold value), then we can say that for this point cloned code is getting about that proportion of changes which it should get considering its proportion in the method. This is the ideal case (indicated by $\bigcirc$ in Table XI) which does not indicate any positive or negative impact of cloned code. Such a point falls in Category 3 (NEUTRAL).

(2) If $CPCM > EPCM$ with an eligibility value greater than or equal to the threshold value, we understand that cloned portions of the cloned methods are getting more changes than they would get in the ideal situation. In other words, the instability of cloned methods due to cloned portions is higher than the instability of the cloned methods due to non-cloned portions. Such decision points (marked with $\ominus$) belong to Category 2 (CLONES LESS STABLE).

(3) The decision points (indicated by $\oplus$) where $CPCM < EPCM$ with an eligibility value greater than or equal to the threshold value belong to Category 1 (CLONES MORE STABLE).

We use Eq.23 to calculate the eligibility value.

$$EligibilityValue = \frac{(HVal - LVal) * 100}{LVal} \tag{23}$$

Here, HVal stands for Higher value between $EPCM$ and $CPCM$ where LVal is elaborated as Lower value between these two. An eligibility value of at least 10 is treated as a significant one as was done for some previous cases. In the following four sections we answer the seventh research question RQ 7.

Overall analysis: We draw the graph in Fig. 26 for overall analysis. We see that each clone detector suggests a much higher stability of cloned code compared non-cloned code. In case of the clone detection results of each clone detector

TABLE XI: Average Instability per Cloned Method by Lozano and Wermelinger's methodology

Lang	Systems	Type 1			Type 2			Type 3			NiCad-Combined			CCFinderX-Combined		
		$CPCM$	$EPCM$	Rem	$CPCM$	$EPCM$	Rem	$CPCM$	$EPCM$	Rem	$CPCM$	$EPCM$	Rem	$CPCM$	$EPCM$	Rem
Java	DNSJava	0.8547	0.7639	⊖	0.4187	0.5721	⊕	0.3829	0.5102	⊕	0.3473	0.4967	⊕	0.1939	0.2748	⊕
	Ant-Contrib	0.3888	0.3967	○	0.2307	0.3913	⊕	0.1666	0.5484	⊕	0.2236	0.5346	⊕	0.0769	0.3355	⊕
	Carol	0.3895	0.5778	⊕	0.5181	0.7023	⊕	0.6431	0.6340	○	0.6383	0.6725	○	0.2216	0.3258	⊕
	jabref	0.1177	0.5142	⊕	0.1413	0.4164	⊕	0.1983	0.4734	⊕	0.2123	0.4322	⊕	0.1034	0.2178	⊕
C	Ctags	0.1875	0.2887	⊕	0.2965	0.3147	○	0.2347	0.3235	⊕	0.2704	0.3519	⊕	0.2021	0.3230	⊕
	Camellia	0.0152	0.0812	⊕	0.0745	0.0852	⊕	0.1588	0.3668	⊕	0.1606	0.3422	⊕	0.1562	0.1695	○
	QMail Admin	0.1234	0.128	○	0.0890	0.0755	⊖	0.1844	0.1674	⊖	0.1820	0.1620	⊖	0.11	0.1057	○
	Gnumake Uniproc	0.4044	0.7947	⊕	0	0.0444	⊕	0.0145	0.2918	⊕	0.10	0.6739	⊕	0.1935	0.1904	○
C#	GreenShot	0.5172	0.8223	⊖	0.4418	0.9004	⊕	0.4795	0.8407	⊕	0.5254	0.5851	⊕	0.1932	0.8054	⊕
	ImgSeqScan	0	0.4705	⊕	0	0.8082	⊕	0	0.4876	⊕	0	0.6591	⊕	0.0667	0.1866	⊕
	Capital Resource	0	0.8940	⊕	0	0.7332	⊕	0.80	0.4348	⊖	0.80	0.4635	⊖	0.3225	0.3091	○
	MonoOSC	0.3333	0.2751	⊖	0.60	0.8502	⊕	0.90	0.9352	○	0.8101	0.8578	○	0.2274	0.3253	⊕

$CPCM$= Average percentage of changes taking place to the cloned portions of cloned methods.
$EPCM$= Average percentage of cloning per method. R = Remark
⊕= *$CPCM$ <$EPCM$ (Category 1, CLONES MORE STABLE)* ⊖= *$CPCM$ >$EPCM$ (Category 2, CLONES LESS STABLE)*
○= *The decision point falls in Category 3*

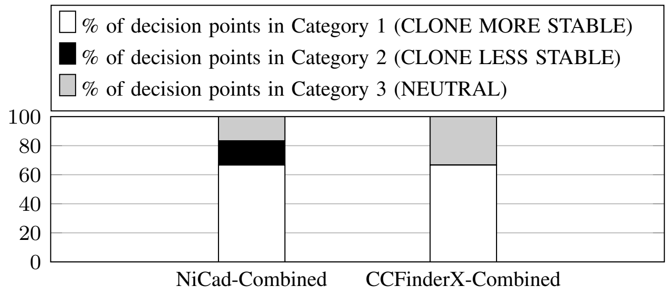


Fig. 26: Overall analysis for average instabilities of cloned methods.

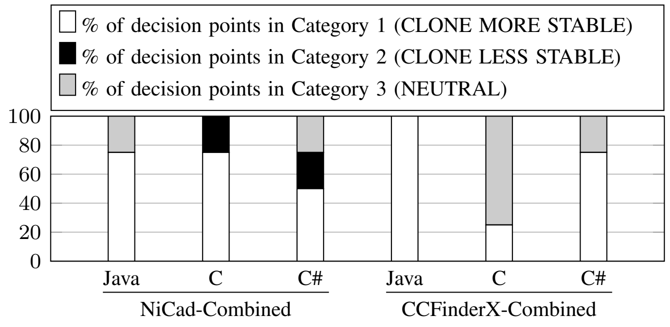


Fig. 27: Language centric statistics for average instabilities of cloned methods

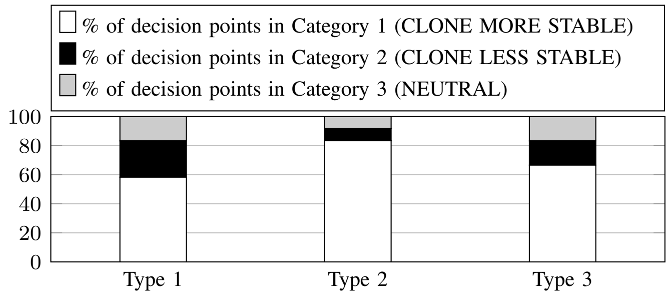


Fig. 28: Type centric statistics for average instabilities of cloned methods.

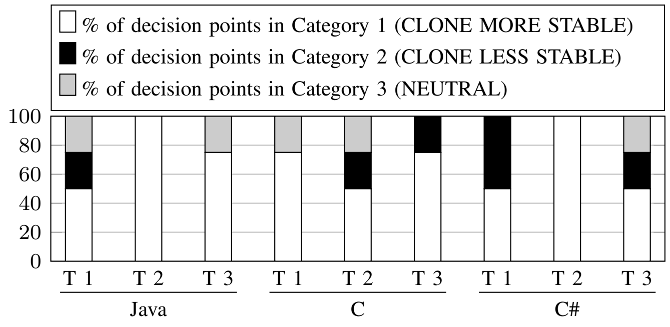


Fig. 29: Language-wise type centric statistics for average instabilities of cloned methods

(NiCad and CCFinderX), around 66.67% of the decision points belong to Category 1 (CLONE MORE STABLE).

Language centric analysis: The language centric analysis presented in the graph of Fig. 27 constructed from Table XI suggests that *for each of the programming languages considering each clone detector, cloned code introduces less instability to a software system than the instability introduced by non-cloned code*.

Type centric analysis: According to the type centric statistics in the graph of Fig. 28, *the instability of the cloned methods due to each clone-type is less than the instability of the cloned methods due to non-cloned code.*

Type Centric Analysis for Each Language: Also, in case of type centric statistics for each candidate programming language (Fig. 29 constructed from Table XI) we see that *no clone types are notably unstable for the maintenance phase compared to non-cloned code*. Thus, this analysis agrees with the previous analyses regarding this metric.

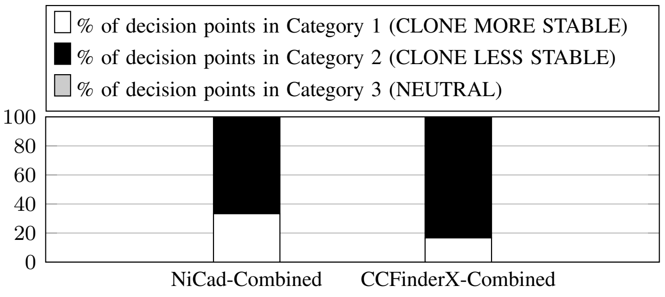


Fig. 30: Overall analysis for change dispersion.

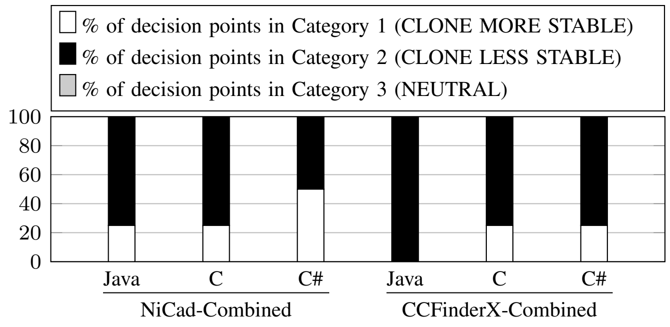


Fig. 31: Language centric statistics for change dispersion

Answer to RQ 7: According to our analysis on *average instability per cloned method*, we see that code clones are more stable than non-cloned code. However, this metric considers the changes in the clone and non-clone portions of the cloned methods. It ignores the changes in the fully non-cloned methods. Possibly, this is the reason why our finding regarding this metric contradicts with our findings regarding the metrics: *modification probability*, *impact*, and *likelihood*. Our finding regarding *average instability per cloned method* also contradicts with the finding from the original study [36] performed by Lozano and Wermelinger. They used CCFinderX in their study for detecting code clones. We also use CCFinderX with the same settings. However, their study was performed only on five systems written in Java. We perform our study on 12 subject systems covering three programming languages (Java, C, and C#). They considered only Type 1 and Type 2 clones in their study. We consider all three types of clones (Type 1, Type 2, and Type 3) in our study. Possibly, these are the reasons behind the contradiction.

G. Analysis of the Experimental Results Regarding Change Dispersion

Using our proposed methodology [42], we calculated the change dispersions in cloned (CD_c) and non-cloned code (CD_n) according to the equations: Eq. 20 and Eq. 21 respectively. However, these equations provide us percentages. We normalized these values within zero to one by dividing them by 100. The normalized dispersions are shown in Table XII. In the following four paragraphs we answer the eighth research question RQ 8 regarding change dispersion.

Overall analysis: According to our overall analysis demonstrated in Fig. 30, each of the two clone detectors (NiCad and CCFinderX) suggests a higher dispersion of changes in cloned code than in non-cloned code for most of the decision points. Regarding NiCad-Combined results we see that 66.67% of the decision points belong to Category 2 (CLONE LESS STABLE). This percentage for CCFinderX-Combined results is 83.33%. Thus, *changes in the cloned portions of a subject system are more scattered than the changes in the non-cloned portions*. In other words, *the proportion of methods affected by the changes in cloned code is generally greater than the proportion of methods affected by the changes in the non-cloned code.*

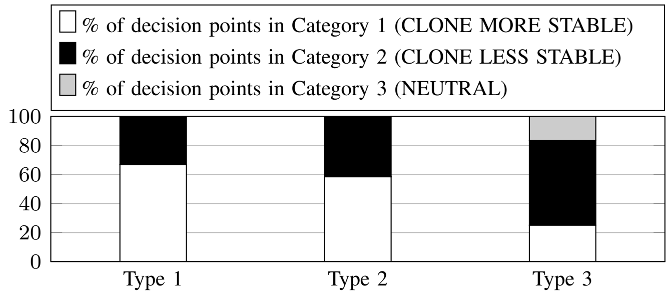


Fig. 32: Type centric statistics for change dispersion.

Language centric analysis: From the graph in Fig. 31 we see that each clone detector suggests a higher dispersion of changes in cloned code than in non-cloned code for each language except C# under NiCad-Combined case. From such a scenario we can decide that *dispersion of changes in code clones is generally higher compared to the dispersion of changes in non-cloned code.*

Type centric analysis: According to the type centric statistics shown in the graph of Fig. 32 (constructed from Table XII), *Type 3 clones are more likely to get highly dispersed changes compared to the other two types (Type 1, Type 2) of clones. While Type 3 clones appear to be more unstable than non-cloned code, the other two types appear to be more stable.*

Type Centric Analysis for Each Language: According to the language-wise type centric statistics shown in graph of Fig. 33 we see that Type 3 clones in Java and both of the

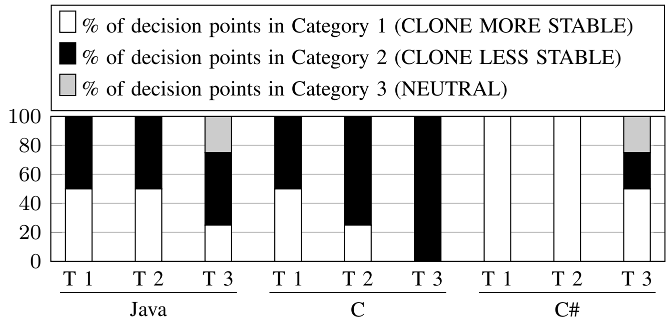


Fig. 33: Language-wise type centric statistics for change dispersion

TABLE XII: Change Dispersion (CD) in Cloned and Non-cloned Methods

Lang	Systems	Type 1			Type 2			Type 3			NiCad-Combined			CCFinderX-Combined		
		CD_c	CD_n	Rem	CD_c	CD_n	Rem	CD_c	CD_n	Rem	CD_c	CD_n	Rem	CD_c	CD_n	Rem
Java	DNSJava	0.2453	0.0591	⊖	0.1517	0.0782	⊖	0.1816	0.0755	⊖	0.17	0.06	⊖	0.18	0.07	⊖
	Ant-Contrib	0.1764	0.0163	⊖	0.0222	0.0195	⊖	0.05	0.0197	⊖	0.0526	0.0176	⊖	0.05	0	⊖
	Carol	0.0587	0.1950	⊕	0.0935	0.1933	⊕	0.1892	0.1972	○	0.2056	0.0912	⊖	0.23	0.07	⊖
	jabref	0.1123	0.2043	⊕	0.0895	0.2178	⊕	0.1305	0.1844	⊕	0.1234	0.1985	⊕	0.31	0.08	⊖
C	Ctags	0	0.1004	⊕	0.20	0.0966	⊖	0.1453	0.0978	⊖	0.13	0.09	⊖	0.21	0.10	⊖
	Camellia	0	0.0985	⊕	0.125	0.0955	⊖	0.35	0.0876	⊖	0.31	0.08	⊖	0.30	0.09	⊖
	QMail Admin	0.50	0.0729	⊖	0.4285	0.0803	⊖	0.60	0.0815	⊖	0.53	0.07	⊖	0.50	0.12	⊖
	Gnumakeuniproc	0.125	0.0042	⊖	0	0.0050	⊕	0.0138	0.0051	⊖	0.04	0.06	⊕	0.0126	0.0273	⊕
C#	GreenShot	0.0888	0.2932	⊕	0.2296	0.2949	⊕	0.3037	0.3047	○	0.1088	0.0388	⊖	0.14	0.05	⊖
	ImgSeqScan	0.0	0.0376	⊕	0.0	0.0373	⊕	0.0	0.0372	⊕	0.0	0.0372	⊕	0.25	0.0055	⊖
	Capital Resource	0.0	0.0492	⊕	0.0	0.0479	⊕	0.0395	0.0447	⊕	0.0395	0.0447	⊕	0.0395	0.0458	⊕
	MonoOSC	0.0317	0.1048	⊕	0.0526	0.1042	⊕	0.3913	0.1003	⊖	0.34	0.10	⊖	0.3944	0.12	⊖

CD_c= Change Dispersion in Cloned Code. *CD_n= Change Dispersion in Non-cloned Code.* *Rem = Remark*
⊕= *$CD_c < CD_n$ (Category 1, CLONES MORE STABLE)* ⊖= *$CD_c > CD_n$ (Category 2, CLONES LESS STABLE)*
○= *The decision point falls in Category 3*

Type 2 and Type 3 clones in C appear to be more unstable than non-cloned code during software maintenance. However, each of the three clone-types of C# seems to be more stable compared to non-cloned code.

Answer to RQ 8: According to our overall analysis considering the combined-type results, *changes in code clones are generally more dispersed (i.e., scattered) compared to the changes in non-cloned code*. Such a finding complies with the finding from the original study [42].

We also performed language centric and type centric analysis for *change dispersion*. Our language centric analysis (Fig. 33) demonstrates that *code clones in Java and C systems have higher likeliness of getting more dispersed changes than non-cloned code in these systems. The opposite is true for the code clones in C# systems.* According to our type centric analysis, *Type 3 clones have a higher possibility of experiencing more dispersed changes than non-cloned code compared to the other two clone-types (Type 1 and Type 2).*

H. Replication of the Original Studies

We have replicated the original studies using our uniform framework. For the purpose of replication we used the same clone detection tools and subject systems as of the original studies. The purpose of replication was to determine whether our implemented uniform framework produces experimental results that are equivalent to the results of the original studies. If the results are equivalent, then we realize that we can reasonably depend on our uniform framework for investigating clone stability. In the following paragraphs we describe the findings of our replicated experiments with original setup.

Replication of the study of Hotta *et al.*: We replicated the original study of Hotta *et al.* [24] using Simian [59] clone detection tool. We used the default configurations for Simian as in the original study. This setting considers six lines of code as minimum clone size. We measured the modification frequencies of cloned (MF_c) and non-cloned (MF_n) code for the same subject systems used in the original study. Table XIII shows the modification frequencies of cloned and non-cloned code for the systems considering all types of clones. For

TABLE XIII: Modification Frequencies by Hotta et al.'s Methodology with Original Setup

	Systems	Simian-Combined			
		MF_c	MF_n	Remark	#Revisions
Java	Adserverbeans	32.0335	60.0428	⊕	125
	DatabaseToUML	2.9124	11.8236	⊕	60
	EclEmma	15.7983	28.8905	⊕	1736
	Freecol	6.1282	14.6388	⊕	6000
	OpenYmsg	27.4103	16.8265	⊖	304
	Squirrel	9.8247	19.3574	⊕	6737
	Threecam	4.0774	10.7737	⊕	14
C++	FileZilla	10.7072	12.2153	⊕	7634
	Gamescanner	23.6345	22.3518	⊖	457
	Tritonn	77.8368	113.9316	⊕	100
	Winmerge	5.1693	12.7347	⊕	7618
C	QMail Admin	66.4018	73.376	⊕	317

MF_c= Modification Frequency of Cloned Code
MF_n= Modification Frequency of Non-Cloned Code
⊕= *$MF_c < MF_n$ (Category 1, CLONES MORE STABLE)*
⊖= *$MF_c > MF_n$ (Category 2, CLONES LESS STABLE)*

all the subject systems except Gamescanner and QmailAdmin our findings agree with the overall conclusion of Hotta *et al.* that modification frequencies for cloned code are less than the modification frequencies of non-cloned code. These imply that cloned code is more stable than non-cloned code. Due to some small variations in the experimental settings such as the number of revisions analyzed, the values of the metrics are not comparable by absolute values. However, our conclusions for subject systems regarding the modification frequencies of cloned and non-cloned code agree with the findings of the original study in most cases.

The original study of Hotta *et al.* used clone detection tools CCFinder, CCFinderX, Simian and Scorpio [64]. For evaluation of our implementation in original settings, we opted for clone detection tools Simian and Scorpio as those tools were not used in our studies reported in Section VII-A. From our investigation on clone detection results from Scorpio we observed that Scorpio detects partially-overlapped and fully-overlapped code blocks as clones unlike all other tools we used in our study. However, our implementation do not consider overlapped clones and we do not have information on how it was handled in Hotta *et al.*'s original study.

Our finding from the replication of the study of Hotta et al.: According to our replication study, cloned code generally appears to be more stable (i.e., less frequently modified) than non-cloned code. Such a finding agrees with the finding from the original study.

TABLE XIV: Modification Probability by Göde et al.'s Methodology with Original Setup

		iClones-Combined			
	Systems	MP_c	MP_n	Remark	#Revisions
Java	ArgoUML	0.000166443	0.000153663	⊖	15000
	Squirrel	0.000093629	0.000095191	⊕	6737

MP_c= Modification Probability of Cloned Code
MP_n= Modification Probability of Non-Cloned Code
⊕= $MP_c < MP_n$ (Category 1, CLONES MORE STABLE)
⊖= $MP_c > MP_n$ (Category 2, CLONES LESS STABLE)

Replication of the study of Göde and Harder : We replicated the study by Göde and Harder [19] with the same subject systems and the clone detection tool iClones [63] used in the original study. We used the default configuration settings for iClones to detect clones of all Types (Type 1, Type 2 and Type 3) for a minimum size of 100 tokens. We measure modification probabilities for cloned (MP_c) and non-cloned (MP_n) code for the subject systems ArgoUML and Squirrel used in the original study. Our results in Table XIV show that for Squirrel the modification probability of cloned code is lower than the modification probability of non-cloned code. This implies that cloned code is more stable than non-cloned code which agrees with the original findings by Göde and Harder. However, for ArgoUML the modification probability of cloned code is slightly higher than the modification probability of non-cloned code. This finding for ArgoUML differs from the original study. We have used the most updated version of iClones available which is different from the version used in the original study. Any difference in clone detection results may have significant impact on the stability analysis as mentioned by different existing studies [24], [36], [37].

Finding from our replication of the study of Göde and Harder : While the original study conducted by Göde and Harder reports that cloned code is more stable (i.e., cloned code has less modification probability) than non-cloned code, our replication study result partially agrees with this finding. The original study investigated two subject systems and both of these systems indicated higher stability of cloned code. However, in our replication study, one of these two systems indicate higher instability of cloned code compared to non-cloned code.

Replication of the study of Krinke: To evaluate metric for our implementation of Krinke's [33] methodology we analyzed the same subject systems and used the text-based clone detection tool Simian as in the original study. We used the default settings for Simian with minimum clone size of 6 lines. For ArgoUML we analyzed revision 19915 (18995 in original study) and for JEdit we analyzed revision 24443 (19285 in original study). Krinke's methodology applies SVN *blame* command on the last revision of a system to retrive the last change dates for source code lines. Thus we have analyzed updated revisions than the revisions used in the original studies. Our experimental results for ArgoUML and JEdit are shown in Table XV. Our results show that for ArgoUML the average last change date ($ALCD_c$) for cloned code (5-JUL-2007) is newer than the average last change date (1-APR-2007) for non-cloned code ($ALCD_n$). This implies that clones are younger than non-clone code. This finding agree with the original study. However, for JEdit the average last change date for cloned code (21-JUL-2006) is newer than the average last change date (3-JUN-2006) for non-cloned code. This finding do not agree with the conclusion of the original study by Krinke. For JEdit we analyzed a considerable number of more revisions which might have introduced differences in our findings for JEdit.

TABLE XV: Average Last Change Dates by Krinke's methodology with original setup

		Simian-Combined			
	Systems	$ALCD_c$	$ALCD_n$	Remark	Revision
Java	ArgoUML	5-JUL-2007	1-APR-2007	⊖	19915
	JEdit	21-JUL-2006	3-JUN-2006	⊖	24443

$ALCD_c$= Average Last Change Date of Cloned Code
$ALCD_n$= Average Last Change Date of Non-Cloned Code
⊕= $ALCD_c$ is older than $ALCD_n$ (Category 1, CLONES MORE STABLE)
⊖= $ALCD_c$ is newer than $ALCD_n$ (Category 2, CLONES LESS STABLE)

Finding from our replication of the study of Krinke: According to our replication study, cloned code appears to exhibit a higher instability than non-cloned code. However, such a finding disagrees with the finding from the original study of Krinke.

TABLE XVI: Impact of cloned and non-cloned code by the methodology of Lozano and Wermelinger with original setup

		CCFinderX-Combined			
	Systems	ICC	INC	Remark	#Revisions
Java	Freecol	0.002602425	0.002317386	⊖	6000
	Ganttproject	0.010625655	0.009936604	⊖	2629
	JBoss	0.005176758	0.004745167	⊖	3000

ICC= Impact of Cloned Code
INC= Impact of Non-Cloned Code
⊕= $ICC < INC$ (Category 1, CLONES MORE STABLE)
⊖= $ICC > INC$ (Category 2, CLONES LESS STABLE)

TABLE XVII: Likelihood of changes of cloned and non-cloned methods by the methodology of Lozano and Wermelinger with original setup

		CCFinderX-Combined			
	Systems	LCC	LNC	Remark	#Revisions
Java	Freecol	0.007500568	0.002713309	⊖	6000
	Ganttproject	0.00421473	0.002518339	⊖	2629
	JBoss	0.008984112	0.004929813	⊖	3000

LCC= Likelihood of Cloned Code
LNC= Likelihood of Non-Cloned Code
⊕= $LCC < LNC$ (Category 1, CLONES MORE STABLE)
⊖= $LCC > LNC$ (Category 2, CLONES LESS STABLE)

Replication of the study of Lozano and Wermelinger

for likelihood and impact: We evaluated the metrics for our implementation of the study by Lozano and Wermelinger [37] using subject systems and the same clone detection tool (CCFinderX) used in the original study. Table XVI shows the *impact* of cloned and non-cloned code for three subject systems used in the original study. Our results show that for all three subject systems the impact of cloned code (ICC) is greater than the value of the impacts of non-cloned code (INC). This suggests that clones are less stable than non-cloned code which agrees with the original study by Lozano and Wermelinger. Again, Table XVII shows the *likelihood* of the cloned and non-cloned code. The results show that for all three subject systems, the likelihood of the cloned code ((LCC)) is greater than the likelihood of the non-cloned code ((LNC)). This implies that clones are less stable than non-cloned code which agrees with the original study.

Finding from the replication of the study of Lozano and Wermelinger regarding likelihood and impact: Our replication study implies that cloned code is more unstable than non-cloned code. This finding agrees with the finding from the original study.

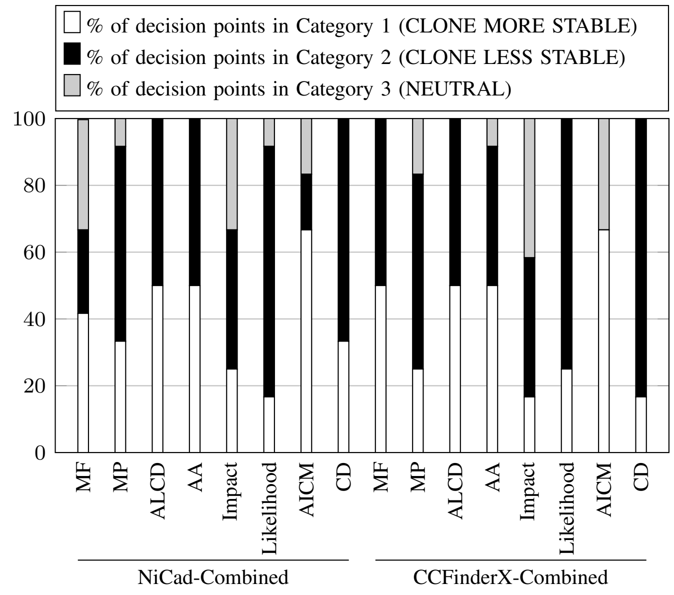
TABLE XVIII: Average Instability per Cloned Method by Lozano and Wermelinger's methodology with original setup

		CCFinderX-Combined			
	Systems	$CPCM$	$EPCM$	Remark	#Revisions
Java	Columba	0.279847182	0.3068405177	⊕	465
	Freecol	0.114571869	0.210005715	⊕	6000
	Ganttproject	0.157217456	0.222673575	⊕	2629
	JBoss	0.201973389	0.278213410	⊕	3000

$CPCM$= *Average percentage of changes taking place to the cloned portions of cloned methods*
$EPCM$= *Average percentage of cloning per method*
⊕= $CPCM < EPCM$ *(Category 1, CLONES MORE STABLE)*
⊖= $CPCM > EPCM$ *(Category 2, CLONES LESS STABLE)*

Replication of the study of Lozano and Wermelinger for stability and extension: To evaluate the metrics for our implementation of the study by Lozano and Wermelinger [36] we used same subject systems and the clone detection tool CCFinderX with same parameter settings. Our experimental results for four Java subject systems used in the original study are shown in Table XVIII. Our results show that the average percentage of changes taking place to the cloned portions of the cloned method ($CPCM$) is smaller than the average percentage of cloning per method ($EPCM$). These results for all four subject systems suggest that clones are more stable. This findings do not agree with the findings from the original study.

These differences in results might be due to the differences in experimental setup. The original study neither clearly specifies the number of revisions considered nor the configuration of the clone detection tools. However, we used the default configuration for the clone tool CCFinderX and the same number of revisions of the systems we used for the other study of Lozano and Wermelinger [37] . In most cases we have analyzed higher number of revisions than we perceive from the result representation in the original study. This includes the analysis of source code of comparatively more mature stage



MF = Modification Frequency MP = Modification Probability
ALCD = Average Last Change Date AA = Average Age
AICM = Average Instability per Cloned Method CD = Change Dispersion

Fig. 34: Proportions of decision points for 8 metrics considering combined type results of each clone detector

of evolution. Again, the original study shows that clones are comparatively less stable at the earlier revisions and clones tend to be more stable over time. Inclusion of more revisions thus might have contributed to our findings to be different from the original study.

From our replication of studies with original setups we observe that our results are mostly equivalent to the results of the original studies. Our results differ in few cases from the original findings. As the subject systems and tools used in the original studies are continuously evolving, it is possibly impractical to expect absolutely exact results to reproduce. However, the equivalent results in most cases imply that our uniform framework reliably integrates the existing methodologies for stability analysis of code clones.

Finding from the replication of the study of Lozano and Wermelinger regarding stability and extension: Our replicated study disagrees with the finding from the original study and indicates that cloned code is more stable than non-cloned code.

VIII. Cumulative Statistics and Analysis of Metrics

So far we have presented our four-dimensional analysis for the results obtained for individual metrics. This section presents our analysis of experimental results from different perspectives by aggregating all the eight metrics (of seven methodologies).

A. Overall analysis

We performed our overall analysis in the following way considering the combined type results. We draw a graph in Fig. 34 that accumulates all overall statistics of all the eight metrics considering **NiCad-Combined** and **CCFinderX-Combined** results.

Analyzing the NiCad-Combined results: Let us consider the bars corresponding to NiCad-Combined results in Fig. 34. We see that according to four metrics: (i) MP (modification probability), (ii) Impact (impact of cloned or non-cloned code), (iii) Likelihood (likelihood of changes in cloned or non-cloned code), and (iv) CD (change dispersion) cloned code appears to exhibit higher instability compared to non-cloned code. In each of these cases, the percentage of decision points in Category 2 (CLONE LESS STABLE) is higher than the percentage in Category 1 (CLONE MORE STABLE). In the cases of the metrics: ALCD (average last change date) and AA (average age), 50% of the decision points belong to both Category 1 and Category 2. The remaining two metrics: MF (modification frequency) and AICM (average instability of cloned methods) suggest higher stability of cloned code compared to non-cloned code. We finally see that according to the combined-type results of NiCad clone detector, *majority of the metrics suggest higher instability of cloned code than non-cloned code*.

Wilcoxon Signed Rank Tests for NiCad-Combined results: We perform Wilcoxon Signed Rank (WSR) Tests [3], [4] considering the combined type results of NiCad to determine whether there is a statistically significant difference between the metric values of cloned and non-cloned code. In a table that contains the decision points for a particular metric (e.g., Table V), we get 12 decision points from 12 subject systems under the heading **NiCad-Combined**. From these 12 decision points we obtain 12 metric values for cloned code and 12 corresponding values for non-cloned code. The metric values for cloned and non-cloned code are paired. We apply Wilcoxon Signed Rank Test [3], [4] on these 12 paird samples to determine whether the 12 metric values for cloned code are significantly different than the 12 metric values for non-cloned code. Table XIX shows the test results for seven metrics (i.e., excluding *average last change date*). We exclude *average last change date* from consideration, because this metric provides dates. We cannot apply Wilcoxon Signed Rank Tests on dates. However, we mentioned that the *average age* metric is a variant of the *average last change date* metric. As we perform tests for *average age*, we believe that exclusion of *average last change date* from consideration will not affect our findings. We should note that the WSR test is non-parametric, and thus, it does not require the samples to be normally distributed [3]. We perform our tests considering a significance level of 5%.

In Table XIX under **NiCad-Combined** heading, the *p-value* (Probability value) regarding modification frequency (MF) is 0.27134. As the *p-value* is greater than 0.05, the 12 modification frequencies of cloned code from 12 subject systems are not significantly different than the corresponding modification frequencies of non-cloned code. We see that *p-value* is greater than 0.05 for most of the cases under **NiCad-Combined** heading. In the case of *change dispersion*, the *p-value* is 0.05. As the *p-value* is not smaller than 0.05, test result is not significant in this case too. We see that only in the case of AICM (average instability per cloned method), the

TABLE XIX: Wilcoxon Signed-Rank tests for the metrics

	NiCad-Combined	**CCFinderX-Combined**
MF	p = 0.27134	p = 0.58232
MP	p = 0.63836	p = 0.4354
AA	p = 0.93624	p = 0.87288
Impact	p = 0.5287	p = 0.4354
Likelihood	p = 0.13622	p = 0.15854
AICM	p = 0.0278, r = 0.4483	p = 0.01208, r = 0.5124
CD	p = 0.05	p = 0.0048, r = 0.5764

MF = Modification Frequency MP = Modification Probability
ALCD = Average Last Change Date AA = Average Age
AICM = Average Instability per Cloned Method
CD = Change Dispersion p = Probability r = Effect Size

p-value (0.0278) is less than 0.05. However, the effect size (r = 0.4483) is not large (i.e., $r < 0.5$) [6]. Moreover, the AICM metric disregards the changes occurred in the fully non-cloned methods. Thus, we cannot rely only on this metric to compare the stability of cloned and non-cloned code. Finally, statistical significance tests on the combined type results of NiCad cannot help us determine whether there is any significant difference between the stabilities of cloned and non-cloned code.

Analyzing the CCFinderX-Combined results: We now consider the overall statistics (Fig. 34) corresponding to the combined-type results of CCFinderX. Interestingly, the same four metrics (as we obtained by analyzing NiCad-Combined results): (i) MP (modification probability), (ii) Impact (impact of cloned or non-cloned code), (iii) Likelihood (likelihood of changes in cloned or non-cloned code), and (iv) CD (change dispersion) suggest higher instability of code clones compared to non-cloned code. According to two metrics: AA (Average age), and AICM (Average instability of cloned methods) cloned code appears to be more stable. For the remaining two metrics, MF (modification frequency) and ALCD (average last change date), 50% of the decision points belong to both Category 1 and Category 2. We again see that *majority of the metrics suggest higher instability of cloned code than non-cloned code*.

We should note that the combined-type results of NiCad and CCFinderX clone detector are not equivalent. While NiCad results contain Type 3 clones, CCFinderX does not detect Type 3 clones. Moreover, the underlying detection techniques of these two clone detectors are different. NiCad is a text similarity based clone detector. CCFinderX detects clones on the basis of token similarity. However, even with these differences in detection techniques and detection results, the overall scenarios of the comparative stability of cloned and non-cloned code implied by these two clone detectors are similar. The same four metrics indicate higher instability of cloned code according to each clone detector.

Let us consider the first metric, modification probability (MP), of the four metrics that indicate higher instability of code clones. According to the implication of this metric mentioned in Table I, the proportion of tokens that get affected in the clone regions per commit operation is generally higher compared to the proportion of affected tokens in the non-clone regions. According to the second metric, impact, we realize that changes in a cloned method generally have a higher

TABLE XX: Comparison Between Original Findings and Our Findings Regarding the Candidate Metrics

Investigated Metric	**Original Findings**	**Our Findings using NiCad**	**Our Findings using CCFinderX**
Modification Frequency (MF) (Hotta et al. [24])	⊕	⊕	○
Modification Probability (MP) (Göde and Harder [19])	⊕	⊖	⊖
Average Last Change Date (ALCD) (Krinke [33])	⊕	○	○
Impact (Lozano and Wermelinger [37])	⊖	⊖	⊖
Likelihood (Lozano and Wermelinger [37])	⊖	⊖	⊖
Average Instability of Cloned Methods (AICM) (Lozano and Wermelinger [36])	⊖	⊕	⊕
Average Age (AA) (Mondal et al. [41])	⊖	○	⊕
Change Dispersion (CD) (Mondal et al. [42])	⊖	⊖	⊖

⊕ = *Instability of cloned code is less than that of non-cloned code*
⊖ = *Instability of cloned code is higher than that of non-cloned code*
○ = *Instabilities of cloned and non-cloned code are the same*

impact (i.e., higher number of other methods gets affected as a result) compared to the changes in a non-clone method. The third metric, likelihood, indicates that cloned methods exhibit higher likeliness of getting changed compared to the non-cloned methods during evolution. According to the implication of the fourth metric, change dispersion (CD), changes in the clone regions of a code-base are generally more scattered compared to the changes in the non-clone regions. From our discussions we realize that:

- Code clones generally have a higher likeliness of getting changed during evolution compared to non-cloned code.
- Impact of changing code clones is generally higher compared to the impact of changing non-cloned code.
- Changes in code clones are more scattered compared to the changes in non-cloned code.

Wilcoxon Signed Rank Tests for CCFinderX-Combined results: From the Wilcoxon Signed Rank tests [3], [4] under **CCFinderX-Combined** heading of Table XIX we see that the test result is statistically significant for two metrics: AICM (*average instability of cloned methods*), and CD (*change dispersion*). However, as we noted before, AICM disregards changes that occurred in fully non-cloned methods, and thus, we can rely on it for making stability decision. If we now consider the *change dispersion* metric, we see that the test result is significant with a large effect size of 0.5764. Now, if we look at the decision points under **CCFinderX-Combined** category of Table XII, we realize that most of the decision points (i.e., 10 points out of 12) belong to Category 2 (CLONE LESS STABLE). Thus, according to our significance test regarding *change dispersion*, cloned code can be significantly more unstable than non-cloned code. In other words, the changes in cloned code can be significantly more dispersed than the changes in non-cloned code.

Contrasting our findings with those of the original studies: Table XX shows the findings of the original studies and our studies at a glance. We can see the eight candidate metrics in this table. Most of the findings indicate higher instability of code clones. From the table we realize that our findings using NiCad clone detector are not contradictory to our findings using CCFinderX. For six metrics (*modification probability*, *average last change date*, *impact*, *likelihood*, *average instability per cloned method*, and *change dispersion*) we get the same overall decisions from both clone detectors. For the remaining two metrics (*modification frequency*, and *average age*), the findings are not really contradictory in the sense that none of the clone detectors suggest code clones to be more unstable for each of these two metrics.

We now focus on the original findings. We see that for three metrics (*modification probability*, *average instability per cloned method*, and *average age*), our findings contradict with the original findings. Let us first consider *modification probability*. The original study performed by Göde and Harder [19] was conducted on two Java systems only. Also, the clone detection tool which was used in this study is different from our tools. These might be the possible reasons behind this contradiction. We have also replicated the original study (described in Section VII-H) in the systematic way (by using the subject systems, clone detection tool, and tool settings used in the original study) and found that our implemented uniform framework produces experimental results which are equivalent to the results in the original study. With such a finding we realize that the clone detection tool as well as the tool settings play an important role in determining the comparative stability scenario of cloned and non-cloned code.

Secondly, for calculating *average instability per cloned method*, Lozano and Wermelinger [36] used only CCFinderX for detecting clones and the experiment was performed on five Java systems only. We performed our experiment using both NiCad and CCFinderX on a different set of subject systems covering three programming languages (Java, C, and C#). We consider all three types of clones (Type 1, Type 2, and Type 3) in our study. Lozano and Wermelinger [36] did not consider Type 3 clones. These can be the possible reasons behind the difference between the findings of our study and the original study. From our systematic replication of the original study (discussed in Section VII-H) we find that our replication study does not agree with the original one. We discussed the reason behind this disagreement in Section VII-H.

Thirdly, we discuss the contradiction regarding *average age* metric. In the original study [41] Mondal et al. found that code clones are generally more unstable than non-cloned code. They considered only individual type results of NiCad in their study. We see that our finding using the combined-type results of NiCad is not necessarily contradictory to the finding of Mondal et al. [41]. However, our finding regarding CCFinderX contradicts with Mondal et al.'s [41] finding. The primary reason behind this contradiction is the difference between the clone detection results of NiCad and CCFinderX. Mondal et al. [41] detected code clones using NiCad and investigated each of the three type of clones (Type 1, Type 2, and Type 3) separately. However, CCFinderX does not report clone results by separating them by clone types. Also, it cannot detect Type 3 clones.

Finally, we can say that although there are a few contra-

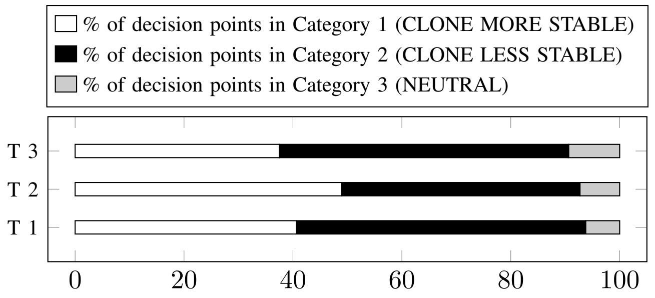


Fig. 35: Proportions of decision points for each clone type considering 8 metrics.

dictions between our findings and the original findings, the three columns of findings (original findings, our findings using NiCad results, our findings using CCFinderx results) in Table XX indicate that *code clones generally have a higher tendency of being more unstable than non-cloned code*.

B. Type centric analysis

We perform type centric analysis on the individual type results. The graph in Fig. 35 shows the comparative instability of the three clone-types considering all three programming languages and 8 metrics. In a particular decision table corresponding to a particular metric a particular clone type (Type 1, Type 2, or Type 3) contributes 12 decision points. So, 12 × 8 = 96 decision points are contributed by each clone type in aggregate. The proportions of the decision points belonging to Category 1, Category 2, and Category 3 for each type is shown in this graph.

From Fig. 35 we see that most of the decision points regarding Type 1 case (i.e., 53.125% of the decision points) suggest higher instability of cloned code. A similar scenario is also observed for Type 3 case. However, we observe an opposite scenario for Type 2 case. The percentage of decision points belonging to Category 1 (CLONE MORE STABLE) for this case is higher compared to the percentage of decision points in Category 2 (CLONE LESS STABLE). Thus, according to the graph (i.e., Fig. 35), *both Type 1 and Type 3 clones appear to be more unstable than Type 2 clones during software maintenance and evolution.*

Fisher's Exact Test: We wanted to determine whether the stability scenario of Type 2 clones is significantly different than that of Type 1 clones or Type 3 clones. For this purpose, we perform Fisher's Exact Test [18] considering the exact count of decision points belonging to Category 1 (CLONE MORE STABLE) and Category 2 (CLONE LESS STABLE) for Type 1, Type 2, and Type 3 cases. We conducted our tests considering a significance level of 5%. The test details are shown in Table XXI. Each of the *p-value*s is greater than 0.05, and thus, we realize that *the stability scenario of Type 2 clones is not significantly different than the stability scenarios of Type 1 and Type 3 clones.* From the graph (Fig. 35) we see that *each of the three types of clones can often be more unstable than non-cloned code.* Managing these three types of code clones through tracking or refactoring can help us minimize maintenance effort and cost of a software system.

TABLE XXI: Fisher's Exact Tests for clone types

	Type 1	Type 2		Type 2	Type 3
Category 1	39	47		47	36
Category 2	51	42		42	51
	P = 0.2326			P = 0.1347	

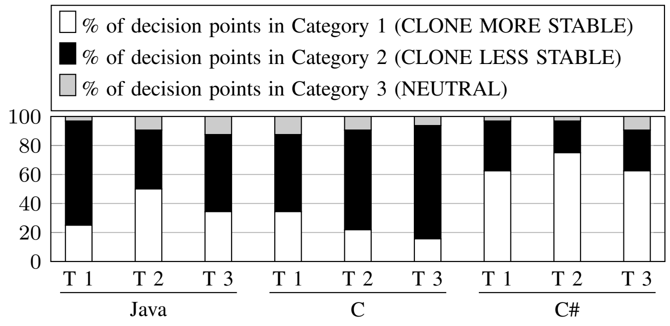


Fig. 36: Proportions of decision points for three clone types of each language considering 8 metrics

According to the discussion, we suggest that programmers should be careful while code cloning, specially, while creating exact clones (Type 1 clones) and Type 3 clones. The possibility that Type 1 or Type 3 clones will be more unstable than non-cloned code is higher than the possibility that Type 2 clones will be more unstable than non-cloned code (Fig. 35). Exact clones can be easily avoided by refactoring which involves two activities: (i) creation of a method containing the Type 1 clone fragment and (ii) removal of all existences of Type 1 clone fragments with proper method calls. But, such straight forward refactoring is not possible for Type 3 clones because such clones contain non-cloned fragments within clone fragments. If we can avoid Type 1 clones from the very beginning of software development by the refactoring activities mentioned above, we can surely reduce the number of Type 3 as well as Type 2 clones to a considerable extent, because Type 2 and Type 3 clones can be created from Type 1 clones.

C. Type centric analysis for each programming language

The graph in Fig. 36 shows the proportions of the decision points for each clone type and each programming language considering eight metrics. When we consider eight metrics, each clone type contributes 32 decision points (four for each combination of clone-type and programming language) for a particular language. The graph (Fig.36) shows the proportions of the decision points (belonging to Category 1, Category 2, and Category 3) of these 32 points for each clone type of a language.

This graph conveys more specific information about the instabilities of three clone types for different languages. According to this graph , *each of the three clone types for C exhibits higher instability compared to non-cloned code in the maintenance phase. In case of Java, Type 1 and Type 3 clones show higher instability. However, each of the three clone-types of C# appears to be more stable than non-cloned code according to our studies.*

TABLE XXII: Fisher's Exact Tests by clone types for programming languages

	Java						C						C#					
	T 1	**T 2**	**T 1**	**T 3**	**T 2**	**T 3**	**T 1**	**T 2**	**T 1**	**T 3**	**T 2**	**T 3**	**T 1**	**T 2**	**T 1**	**T 3**	**T 2**	**T 3**
Category 1	8	16	8	11	16	11	11	7	11	5	7	5	20	24	20	20	24	20
Category 2	23	13	23	17	13	17	17	22	17	25	22	25	11	7	11	9	7	9
P	0.0342		0.4031		0.2924		0.263		0.0786		0.5321		0.2813		0.7881		0.5634	

T 1 = Type 1 T 2 = Type 2 T 3 = Type 3 P = P-value (or Probability value)

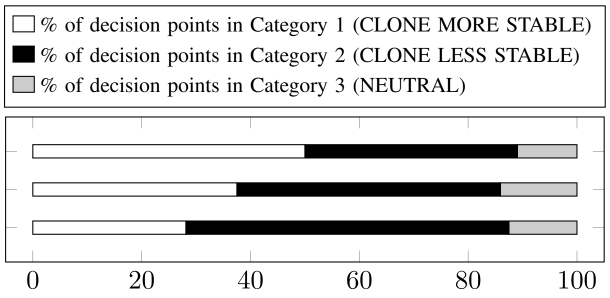


Fig. 37: Proportions of decision points for each programming language considering combined type results of 8 metrics

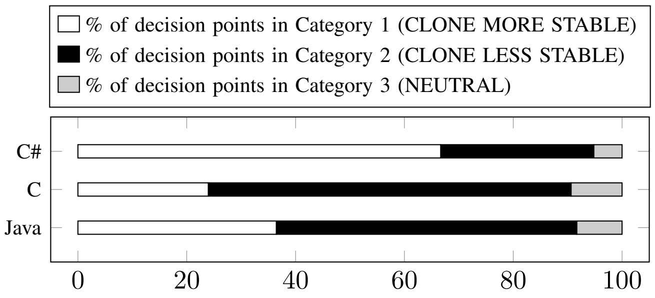


Fig. 38: Proportions of decision points for each programming language considering individual type results of 8 metrics

Fisher's Exact Test: In order to find the validity of the first null hypothesis (regarding RQ 9), we performed Fisher's exact tests [18] for each pair of three clone types (Type 1, Type 2, and Type 3) for each programming language. The test details for Java, C and C# are shown in Table XXII. For the tests, we used the exact counts of the decision points belonging to Category 1 and Category 2 corresponding to each programming language and clone type.

According to the test results, *there is a statistically significant difference between Type 1 and Type 2 clones of Java (p-value = 0.0342 <0.05)*. But, for other two languages, none of the three clone type pairs shows a significant difference.

D. Language centric analysis

We perform language centric analysis considering both combined type results and individual type results. The analysis procedure and observations are described below.

1) Considering combined type results: The graph in Fig. 37 shows the proportions of the decision points of three categories (Category 1, Category 2, Category 3) for each programming language considering eight metrics. We see that each programming language contributes 8 decision points for the combined type cases (NiCad-Combined and CCFinderX-Combined cases) in each table. So, if we consider the combined type results of all the eight tables, a particular language contributes $8 \times 8 = 64$ decision points. This graph shows the proportions of the decision points for each language considering these 64 points.

We see that in case of Java, while only 28.1% (18 decision points) of the decision points belong to Category 1 (CLONES MORE STABLE), 59.4% (38 points) fall in Category 2 (CLONES LESS STABLE). These proportions for C are 37.5% (24 points) and 48.4% (31 points) respectively. However, an opposite scenario is exhibited by C#. While 50% (32 points) of the decision points of this language belong to Category 1, only 39% (25 points) points belong to Category 2. According to this graph, *clones in Java and C are possibly more unstable than the clones in C#.*

2) Considering individual type results: We can see that in each of the tables (corresponding to each of the eight metrics) a particular programming language contributes 12 decision points under the headings, **Type 1**, **Type 2** , and **Type 3**. So, if we consider all of the eight metrics, the number of decision points contributed by a particular programming language is 96. We observe that in case of Java, while only 35 decision points belong to Category 1 (in favor of clones) 53 decision points belong to Category 2 (against clones). The remaining 8 decision points are insignificant. For C, the counts of points belonging to Category 1 and Category 2 are 23 and 64 respectively. The corresponding counts for C# are 64 and 27. We determined the proportions of the decision points belonging to Category 1, 2, and 3 for each language and plotted these in the graph of Fig. 38.

From the graph (Fig. 38) we again see that *clones in both Java and C languages exhibit higher instability compared to the instability exhibited by the clones of C#.*

Fisher's Exact Test: We performed Fisher's exact tests separately for the combined type and individual type cases to validate the null hypothesis 2 regarding the tenth research question (RQ 10). Table XXIII contains the test details for combined type case and Table XXIV contains the test details for individual type case. Each of these tables contains the details of the three tests corresponding to three language pairs. Each test was conducted on the exact counts of the decision points belonging to *Category 1* and *Category 2*. The p-values of the corresponding tests are shown along the last rows of the tables. If the p-value of particular test is less than 0.05, the difference between the observed data for that particular test is significant.

We see that for the combined type case, the test result

TABLE XXIII: Fisher's Exact Tests for prog. languages (Combined type case)

	Java	C		Java	C#		C	C#
Category 1	18	24		18	32		24	32
Category 2	38	31		38	25		31	25
	P = 0.2436			P = 0.0138			P = 0.2567	

TABLE XXIV: Fisher's Exact Tests for prog. languages (Individual type case)

	Java	C		Java	C#		C	C#
Category 1	35	23		35	64		23	64
Category 2	53	64		53	27		64	27
	P = 0.0773			P < 0.0001			P <0.0001	

corresponding to the language pair: Java and C# is significant. The p-value for this pair = 0.0138 (less than 0.05). For the individual type case, the test results for two language pairs: (1) Java and C# and, (2) C and C# are statistically significant because the p-values for these two pairs are less than 0.05.

Thus, we can say that *the instability exhibited by the clones in C# is significantly lower (from our statistical significance tests) than the instability of clones in both Java and C.*

E. Clone detection tool centric analysis

This analysis is based on the combined type results. In each of the eight tables, there are 12 decision points under each of the two headings, **NiCad-Combined** and **CCFinderX-Combined**, corresponding to two clone detection tools, NiCad and CCFinderX. So, the total number of decision points contributed by a particular tool in eight tables is 96 (= 12 x 8). Our tool centric analysis is based on 96 decision points contributed by each tool.

According to the significant decision points belonging to NiCad, 54.8% points (46 points) exhibit higher instability of cloned code than non-cloned code. The remaining 45.2% (38 points) of the significant points contributed by this tool suggest the opposite. These two percentages for CCFinderX are 57.1% (48 points) and 42.9% (36 points) respectively. Thus, *each clone detection tool individually suggests higher instability of cloned code compared to non-cloned code*.

F. System centric analysis

In our system centric analysis we determined the agreement and disagreement of the eight metrics obtained for a particular system and a particular clone case (there are five clone cases in total as indicated in Fig. 1). The agreement-disagreement scenario has been presented in Table XXV. The construction of the table is explained below.

For a particular subject system and a particular clone case

- if majority of the metrics agree with higher instability of cloned code, the corresponding cell in the table is marked with '⊖'.
- if majority of the metrics agree with higher instability of non-cloned code (lower instability of cloned code), the corresponding cell in the table is marked with '⊕'.

TABLE XXV: System centric analysis for five clone cases

Subject systems	T 1	T 2	T 3	NiCad (C)	CCFinderX (C)
DNSJava	⊖	⊖	⊖	⊖	⊖
Ant-Contrib	⊖	⊕	⊖	⊖	○
Carol	○	⊖	⊖	⊖	⊖
Jabref	○	⊕	⊕	⊕	⊖
Ctags	⊖	⊖	⊖	⊕	○
Camellia	⊕	⊖	⊖	⊖	⊖
QMailAdmin	⊖	⊖	⊖	⊖	⊖
GNUMakeUniproc	⊕	⊖	⊖	⊕	⊕
GreenShot	○	⊕	⊖	⊖	⊕
ImgSeqScan	⊕	⊕	⊕	⊕	○
CapitalResource	⊕	⊕	⊕	⊕	⊖
MonoOSC	○	⊕	⊕	⊕	⊖

T1 = Type 1 case of NiCad   T2 = Type 2 case of NiCad
T3 = Type 3 case of NiCad   NiCad(C) = Combined case of NiCad
CCFinderX(C) = Combined case of CCFinderX
⊕ = For a particular subject system and clone case, majority of metrics agree with higher stability of cloned code
⊖ = For a particular subject system and clone case, majority of metrics agree with higher instability of cloned code
○ = For a particular subject system and clone case,the same number of metrics agree with both higher and lower stability of cloned code

- if the number of metrics agreeing with higher instability of cloned code is equal to the number of metrics with lower instability of cloned code, we marked the corresponding cell with '○'.

The Table XXV contains 60 cells where each cell corresponds to a particular subject system and a particular clone case. We have the following observations from this table.

- While 31 cells (51.7%) are marked with '⊖', 22 (36.7%) cells contain '⊕', and the remaining 7 (11.7%) cells contain '○'. Thus, *most of the cells indicate higher instability of cloned code.*
- We calculated the percentages of the cells containing the symbols ⊕ (CLONES MORE STABLE), ⊖ (CLONES LESS STABLE), and ○ (NEUTRAL) for each of the five clone cases and plotted these percentages in the graph of Fig. 39. We see that *for most of the cells belonging to each of the two clone cases: Type 3 (NiCad), and Combined (CCFinderX), majority of the metrics agree with higher instability of cloned code.*
- We also calculated the percentages of the cells containing ⊕, ⊖, and ○ belonging to each of the three programming languages and plotted these percentages in the graph of Fig. 40. We see that*for most of the cells belonging to each of the two programming languages: Java, and C, majority of the metrics agree with higher instability of cloned code.* Such a scenario is also indicated by both Fig. 37 and Fig. 38. However, in Fig. 40, most of the cells belonging to C# agree with higher stability of cloned code.

G. Analysis related to the programming language paradigm

From the combined type results of Table XI we calculated the average EPCMs (average extension of cloning per method) for procedural programming language (C) and object-oriented programming languages (Java and C#) considering both of the two clone detection tools. The average EPCMs are 0.2898,

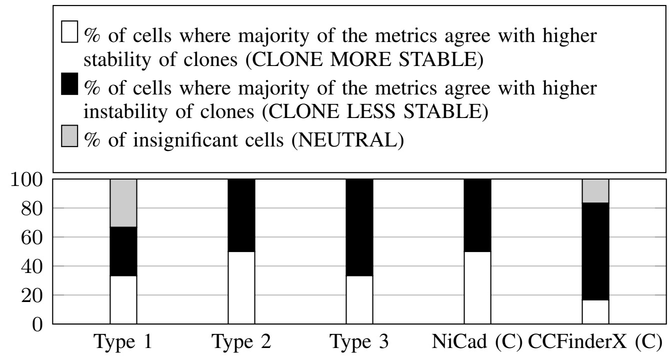


Fig. 39: System centric statistics regarding five clone cases.

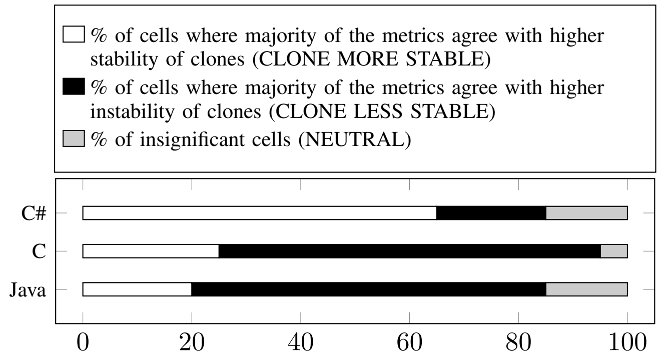


Fig. 40: System centric statistics regarding three programming languages

0.4112 and 0.5239 for C, Java and C# respectively. We observe that *average extension of cloning in C systems is much lower compared to the other two programming languages (Java and C#)*.

We also calculated the averages of CD_cs (Dispersion of Changes in Cloned Code) for the three programming languages from the combined type results of Table XII. These averages are 0.2541, 0.1652 and 0.1640 for C, Java and C# respectively. In this case we see that the changes in the cloned code of C programming language are more dispersed than the changes in the cloned code of the other two programming languages.

From this we infer that *the extension (or proportion) of cloning in procedural programming language (c) is much lower than that of object-oriented programming language (Java and C#) however, the changes to the clones in procedural language are more scattered compared to the changes to the clones in object-oriented languages*.

H. Analyzing the correlation of stability metrics

We wanted to analyze whether there is any correlation among the metrics that we have investigated in our study. Such an analysis can help us identify a compact set of metrics that can be used to realize the stability scenario of the major three clone-types (Type 1, Type 2, and Type 3). We consider the combined-type clone results of NiCad and CCFinderX in this investigation. We excluded the ALCD metric from this investigation, because this metric only provides dates. As we consider the age metric in our investigation, exclusion of ALCD metric does not affect our analysis. Table XXVI shows the analysis result. We used Spearman's Rank Correlation [1], [2] for our analysis. For this correlation, the samples do not need to be normally distributed.

We determine correlation considering each pair of metrics. Let us consider the metrics: MF (modification frequency), and MP (modification probability). Considering the combined type of results of NiCad we get the list of modification frequencies of cloned code from Table V. We also get the corresponding list of modification probabilities from Table VI. We then determine whether these two lists of metric values are correlated or not. We also determine correlation in the same way considering CCFinderX clone results.

Correlation analysis on NiCad-Combined clone results. In our analysis we consider only those cases where the correlation between two metrics is significant. If we look at the column **NiCad-Combined** of Table XXVI, we see that the correlation is significant for five cases: (i) MF and CD, (ii) MP and AA, (iii) MP and CD, (iv) AA and CD, and (v) Impact and Likelihood. In each of these five cases, the *p-value* is less than 0.05. In all other cases, the correlation coefficient is low, and also the correlation is insignificant according to the *p-values* (*p-values* are greater than 0.05). We consider the five cases of significant correlation in our analysis. We see that the metric CD (change dispersion) is positively correlated with three other metrics: MF (modification frequency), MP (modification probability), and AA (average age). Thus, while understanding the stability of cloned code, if one selects the CD (change dispersion) metric, then he/she can possibly exclude MF, MP, and AA from considerations. There is a positive correlation between AA and MP. However, both of these metrics are correlated with CD. Thus, if we consider CD, we do not need to separately consider AA and MP for understanding clone stability. We also see that impact and likelihood have a strong positive correlation. However, these two are not correlated with any other metrics. We also observe that AICM (average instability per cloned method) is not correlated with any other metrics. Finally, according to our analysis on the combined-type clone results of NiCad, the metric set (CD, impact, and AICM) or the set (CD, likelihood, and AICM) can help us realize stability of cloned code.

Correlation analysis on CCFinderX-Combined clone results. We analyzed metric correlations considering CCFinderX clone results in a way which is similar to our analysis using NiCad combined clone results. From the column named **CCFinderX-Combined** of Table XXVI we realize that the metric likelihood has good correlations with AA (average age) and impact. Change dispersion (CD) is positively correlated with modification probability (MP). According to our analysis, the metrics set (MF, CD, likelihood, and AICM) can help us realize the stability of cloned code. However, CCFinderX does not detect Type 3 clones. Thus, our correlation analysis on the CCFinderX clone results only reflects the stability of Type 1 and Type 2 clones. As NiCad detects all three types of clones (Type 1, 2, and 3), we can possibly rely on the correlation analysis on NiCad results to understand the combined stability of these three major clone-types.

TABLE XXVI: Spearman's Rank Correlation among the metrics

	NiCad-Combined	CCFinderX-Combined
MF, MP	r = 0.41958, p = 0.17452	r=0.25175, p=0.42992
MF, AA	r = 0.39161, p = 0.20806	r=0.25874, p=0.41678
MF, Impact	r = 0.48252, p = 0.11211	r=0.12587, p=0.69668
MF, Likelihood	r = 0.34266, p = 0.27557	r = -0.40559, p = 0.19084
MF, AICM	r = -0.18182, p = 0.5717	r = 0.00699, p = 0.98279
MF, CD	**r = 0.63636, p = 0.0261**	r = 0.36364, p = 0.24527
MP, AA	**r = 0.66434, p = 0.01845**	r = 0.08392, p = 0.79541
MP, Impact	r = 0.32867, p = 0.2969	r = 0.16084, p = 0.61752
MP, Likelihood	r = 0.47552, p = 0.11818	r = -0.02098, p = 0.9484
MP, AICM	r = 0.0979, p = 0.76212	r = -0.23077, p = 0.47053
MP, CD	**r = 0.78322, p = 0.00259**	**r = 0.61538, p = 0.03317**
AA, Impact	r = 0.28671, p = 0.36625	r = -0.18182, p = 0.5717
AA, Likelihood	r = 0.3007, p = 0.34226	**r = -0.65035, p = 0.02203**
AA, AICM	r = -0.11189, p = 0.7292	r = -0.2028, p = 0.5273
AA, CD	**r = 0.58741, p = 0.04461**	r = 0.26573, p = 0.40383
Impact, Likelihood	**r = 0.72727, p = 0.00735**	**r = 0.6014, p = 0.03859**
Impact, AICM	r = -0.21678, p = 0.49856	r = -0.54546, p = 0.06661
Impact, CD	r = 0.11888, p = 0.71288	r = -0.08392, p = 0.79541
Likelihood, AICM	r = -0.16783, p = 0.6021	r = -0.1049, p = 0.74561
Likelihood, CD	r = 0.24476, p = 0.44326	r = -0.18881, p = 0.55674
AICM , CD	r = 0.23776, p = 0.4568	r = -0.20979, p = 0.51284

MF = Modification Frequency MP = Modification Probability
ALCD = Average Last Change Date AA = Average Age
AICM = Average Instability per Cloned Method
CD = Change Dispersion p = Probability r = Correlation Coefficient

IX. Discussion

A. Findings

We analyze the comparative stability of cloned and non-cloned code in thousands of revisions of 12 diverse subject systems considering eight stability measurement metrics. According to our analysis we have the following findings.

Finding 1: *Cloned code is often more unstable than non-cloned code during software evolution and maintenance.* More specifically:

- Cloned code has a higher tendency of getting changed compared to non-cloned code (according to our analysis regarding two stability metrics: *modification probability*, and *likelihood*).
- The impact of changing a cloned method (i.e., the number of other methods that might need to be changed as a consequence of changing a cloned method) is generally higher than the impact of changing a non-cloned method (according to our analysis on the *impact* metric).
- Changes in the clone regions of a software system are generally more dispersed compared to the changes in the non-clone regions (according to our analysis on the *change dispersion* metric).

According to our statistical significance tests, changes in cloned code can be significantly more dispersed than the changes in non-cloned code. From such a scenario we believe that code clones can considerably increase software maintenance effort and cost during evolution. Thus, proper management of code clones through refactoring and tracking is necessary with tool support.

Finding 2: *Although not statistically significant, Type 1 and Type 3 clones seem to exhibit higher instability than Type 2 clones.* Minimizing the number of Type 1 clones through refactoring can possibly help us minimize the number of both Type 2 and Type 3 clones, because these two types of clones can be created from Type 1 clones.

Finding 3: *Instability of code clones in Java and C systems is significantly higher (from our statistical significance tests) compared to the instability of code clones in C# systems. Code clones in C# systems are generally more stable than non-cloned code in these systems.* Thus, we should primarily focus on managing code clones in the subject systems written in Java and C. Possibly, we can exclude C# systems from our considerations when taking clone management decisions.

Finding 4: The extension of cloning in procedural programming languages seems considerably smaller than the extension of cloning in object-oriented programming languages. However, changes to the code clones in procedural languages are more dispersed compared to the changes to the code clones in object oriented languages.

Finding 5: According to our correlation analysis, either of the two sets: (*Change Dispersion*, *Impact*, and *AICM*) or (*Change Dispersion*, *Likelihood*, and *AICM*) can help us realize the stability of cloned code of a subject system.

B. Sensitivity analysis regarding the threshold of our calculated eligibility value

In our research we calculate an eligibility value for each decision point where a decision point consists of a metric value for cloned code, and the corresponding metric value for non-cloned code. We also consider a threshold value of 10 such that if the eligibility value for a decision point is greater than or equal to this threshold, then we consider this decision point for decision making purpose; otherwise, we ignore it. We should mention that there is no empirically established threshold value that we could use in our study context. In Section VII, we have explained why we have selected a threshold value of 10 in our study. We have also performed a sensitivity analysis (what-if analysis) for the threshold value in the following way.

Sensitivity Analysis Process. We have eight tables (Table V, VI, VII, VIII, IX, X, XI, XII) in the paper for eight stability metrics. However, for analyzing the decision points in two tables (Table VII and VIII) corresponding to the two metrics, ALCD (Average Last Change Date) and AvgAge (Average Age), we did not calculate an eligibility value by considering the nature of data in these tables. A detailed explanation has been provided in Section VII. As the decision points in Table VII and Table VIII were treated in a different way (i.e., not considering the threshold value) because of the nature of data contained in these tables, we exclude these two tables from our sensitivity analysis. We perform our analysis considering all the decision points (360 points in total) in the remaining six tables. For different values of threshold in the range 0 to 100, we determine the overall stability scenario of cloned and non-cloned code from these 360 decision points. For each threshold

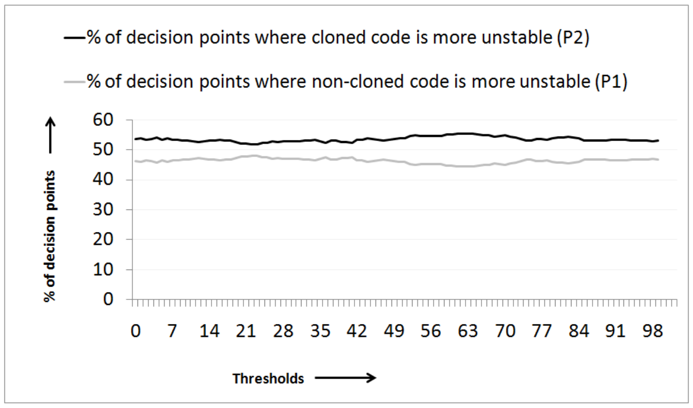


Fig. 41: Sensitivity analysis regarding the threshold for our calculated eligibility value

value, we first identify those decision points that meet the threshold. Considering these decision points, we determine the following two percentages:

- **P1:** The percentage of decision points where cloned code appears to be more stable (i.e., non-cloned code more unstable) with respect to all decision points that meet the threshold.
- **P2:** The percentage of decision points where non-cloned code appears to be more stable (i.e., cloned code is more unstable) with respect to all decision points that meet the threshold.

We plot these two percentages in Fig. 41 for all the threshold values in the range 0 to 100. From the figure we realize that the percentage of decision points where cloned code is more unstable (P2) is always higher than the percentage of decision points where cloned code is more stable (P1). We also see that the two lines demonstrating the two percentages are almost horizontal. It implies that the values of the two percentages remained similar for all the thresholds. From the graph we can decide that the comparative stability scenario of cloned and non-cloned code is independent of the thresholds. In other words, it appears that a threshold value does not impact the stability scenario of cloned and non-cloned code. Thus, our selected threshold of 10 does not affect the reliability of our study findings.

X. Threats to Validity

A. Threats to external validity

The number of subject systems investigated in our study is not sufficient for taking strong decision about the comparative impacts of cloned and non-cloned code. Also, some important factors such as type of developed software, expertise of the responsible programmers, allocated development time etc might have significant effects on cloning and stability of cloned and non-cloned code. We did not consider these factors in our study. However, our selection of subject systems emphasizing on the diversity of application domains, system sizes, implementation languages and the large number of revisions have considerably minimized these drawbacks.

B. Threats to internal validity

For different settings of clone detection tools (NiCad and CCFinderX), the number of detected clones will be different and thus, the comparative stability scenarios of cloned and non-cloned code can also be different. Wang et al. [61] defined this issue as the *confounding configuration choice problem* and conducted an extensive empirical investigation on it to minimize its effects. However, the NiCad settings used in our experiment are considered standard and with these settings NiCad can detect clones with higher precision and recall [52]–[54]. Also, the settings that we have used for CCFinderX are considered equivalent to those of NiCad [39]. Thus, we believe that our findings are important.

Our observation regarding the programming language paradigm is based only on three programming languages. The observation could be more precise with some other programming languages from both procedural and object oriented paradigms. However, as we considered many decision points from both paradigms we expect that our observation cannot be attributed to a chance.

C. Threats to construct validity

All of the metrics investigated in our study are only related to stability (of cloned or non-cloned code). For determining the impact of cloned code on maintenance we should also investigate the relation of clones with bugs, faults and inconsistencies. However, according to our detailed explanation in the introduction, by comparing the instability of cloned code with that of non-cloned code we can determine whether cloned or non-cloned code require higher maintenance effort and cost.

XI. Related Work

We have emphasized clone stability in our research. However, stability is related with some other factors such as harmfulness and longevity. Thus, our related work section not only describes existing research on clone stability, but also discusses the existing work on the harmfulness and longevity of code clones.

A. Clone stability

Hotta et al. [24] studied the stability of clones on software maintenance by determining the modification frequencies of the duplicated and non-duplicated code segments of 15 subject systems. According to their experimental result, the presence of clones does not introduce extra difficulties to the maintenance phase.

Krinke [31] measured how consistently the code clones are changed during maintenance using *Simian* [59] (clone detector) on Java, C and C++ code bases considering Type-I clones only. He found that clone groups changed consistently through half of their lifetime. In another experiment he showed that cloned code is more stable than non-cloned code [32]. In his most recent investigation [33] centered on calculating the average ages of the cloned and non-cloned code, he has shown cloned code to be more stable than non-cloned code by exploiting the capabilities of version controlling system.

In a recent study [22] Harder and Göde replicated and extended Krinke's study [32] using an incremental clone detection technique [20] to validate the outcome of Krinke's study. They supported Krinke by assessing cloned code to be more stable than non-cloned code in general.

Mondal et al. [39] introduced a code stability measure *change dispersion* and compared the change dispersions of cloned and non-cloned code considering 16 subject systems. According to their observation, changes in cloned code are often more dispersed compared to the changes in non-cloned code and thus, cloned code is expected to require higher maintenance effort and cost than non-cloned code. In another study Mondal et al. [40] investigated four code stability metrics on 12 subject systems and found that cloned code is generally less stable than non-cloned code.

Lozano and Wermelinger [37] experimented to assess the effects of clones on the changeability of software using CCFinderX [28] as the clone detector. According to their study, cloned code can sometimes increase the instability of software systems. In another experiment [35], they experienced that cloned code leads to more changes. In their most recent experiment [36] aiming to analyze the imprints of clones over time, they found that cloning introduces higher density of modifications during maintenance.

Kim et al. [30] proposed a model of clone genealogy to study clone evolution. Their study with the revisions of two medium sized Java systems showed that refactoring of clones may not always improve software quality. Saha et al. [56] extended their work by extracting and evaluating code clone genealogies at the release level using 17 open source systems. Their study reports similar findings as of Kim et al. and concludes that most of the clones do not require any refactoring efforts during maintenance. On the other hand, Juergens et al.'s [27] study with large scale commercial systems suggests that inconsistent changes are very frequent to the cloned code and nearly every second unintentional inconsistent change to a clone leads to a fault. Kapser and Godfrey [29] identified different patterns of cloning and experienced that around 71% of the clones could be considered to have a positive impact on the maintainability of the software system. Aversano et al. [8] combined clone detection and modification transactions on open source software repositories to investigate how clones are maintained during the evolution and bug fixing. Their study reports that most of the cloned code is consistently maintained. In another similar but extended study, Thummalapenta et al. [60] indicated that most of the cases clones are changed consistently and for the remaining inconsistently changed cases clones mainly undergo independent evolution.

Chatterji et al. [15] performed a human-based empirical study to resolve the contradiction among the claims regarding the impacts of clones on software maintenance and evolution. Their study includes three surveys each consisting of responses from approximately 20 researchers from the clone research community. From the surveys, the authors feel the necessity of more focused and human-oriented investigations for resolving the contradictions regarding the effects of clones on software maintenance.

We see that while the objective is the same—*determining the impacts of clones on software maintenance*, the researchers considered different approaches with different clone detection tools and subject systems, and finally reported contradictory findings. Our empirical study described in this paper is an attempt to resolve the contradiction using a uniform framework.

B. *Harmfulness of code clones*

Rahman et al. [46] investigated the bug-proneness of cloned and non-cloned code considering four subject systems and found non-cloned code to be more bug-prone than clone code. They investigated only monthly snapshots (i.e., revisions) of the subject systems. Thus, their analysis might exclude some buggy commits from considerations. Selim et al. [58] investigated the fault-proneness of code clones using Cox hazard models. According to their investigation on two subject systems, fault-proneness of code clones might be system dependent.

Wang et al. [62] proposed a Bayesian Networks based machine learning technique to realize the harmfulness of code cloning operations (such as copy/paste activities). They applied their technique on two large scale industrial software systems and found that they could block a considerable proportion of the copy/paste activities that may lead to harmful clones.

Late propagation [43], [45] is a harmful clone evolution pattern. Barbour et al. [10], [11] investigated late propagation and its relatedness with bug-proneness. They introduced eight late propagation patterns and identified the most bug-prone ones. Barbour et al. [10], [11] only considered Type 1 and Type 2 clones in their study. In another study, Mondal et al. [43] investigated the bug-proneness of late propagation considering all three major types of clones: Type 1, Type 2, and Type 3. Existing studies [25], [44] have also investigated bug-replication tendencies and bug densities in three types of code clones.

C. *Longevity of code clones*

Cai and Kim [13] investigated the characteristics of long lived clones. According to their considerations all the code clones in a software system should not be refactored aggressively. Some clones never change during evolution and also, some clones are volatile. Such clones should not be considered for refactoring. According to their investigation, static or spatial characteristics of clones should not be considered when taking clone refactoring decisions. The evolutionary characteristics of clones should be emphasized to identify the important ones for refactoring.

In the previous subsections, we have discussed the existing studies that are directly or indirectly related to clone stability. We have seen that clone stability has been measured in different studies in different ways considering different stability aspects using different experimental settings. In our empirical study, we develop a uniform framework to evaluate all the stability metrics on the same set of subject systems with the same experimental setting. We also evaluated the metrics with the original experimental settings. Through our experimental results, we focus on resolving the contradiction among the existing clone stability studies and determine whether cloned code is more stable than non-cloned code or not. Our study results in interesting findings regarding clone stability.

XII. Conclusion and Future Work

In this empirical study, we implemented seven methodologies and calculated eight stability related metrics using a common framework. We implemented each of these methodologies using two clone detection tools: NiCad and CCFinderX and applied on each of the twelve subject systems written in three programming languages (Java, C, and C#). According to our analysis of the experimental results, *cloned code appears to be more unstable than non-cloned code in general*. Each of the clone detection tools individually suggests cloned code to be more unstable. This scenario disagrees with the already established bias [24], [33] and indicates that *clones are not necessarily stable, and most of the time more unstable than non-cloned code in the maintenance phase*. Thus, clones should be managed with proper tool support. However, although we found cloned code to be generally more unstable than non-cloned code, our statistical significance test results imply that the difference between the instability of cloned and non-cloned code is not statistically significant for most (seven out of eight) of the stability metrics. Only for the stability metric *change dispersion*, cloned code was found to be significantly more unstable than non-cloned code. In other words, *changes occurring in cloned code are significantly more dispersed (i.e., affecting more program entities) than the changes in non-cloned code*. Thus, making changes to a cloned fragment might be riskier than making changes to a non-cloned fragment.

We have systematically replicated (i.e., replicated using their subject systems, clone detection tools, and tool settings) the original studies using our uniform framework. We find that the experimental results produced by our replicated experiments are mostly equivalent to the experimental results of the original studies.

According to our type-centric analysis, Type 1 and Type 3 clones are more unstable compared to the Type 2 clones. Our Fisher's exact test results suggest that Type 1 clones of Java systems are significantly more unstable than Type 2 clones. Possibly, we should give more emphasis on managing Type 1 and Type 3 clones.

Our language centric analysis suggests that clones in Java and C programming languages are more unstable than the clones in C#. Our Fisher's exact test results regarding programming languages imply that code clones in Java and C systems are significantly more unstable compared to the code clones in C# systems. We find that code clones in C# systems mostly exhibit higher stability than non-cloned code in these systems. Thus, when taking clone refactoring decisions we should primarily focus on Java and C systems excluding C# systems from our considerations.

According to our correlation analysis on the eight stability metrics we find that either of the two sets: (Change Dispersion, Impact, and AICM) and (Change Dispersion, Likelihood, and AICM) can be used for realizing the stability scenario of the code clones of a subject system.

Finally, according to our result it seems that object-oriented programming languages promote more cloning than procedural programming languages. However, changes to the clones in procedural programming language appear to be more scattered (i.e., exhibit higher dispersion) compared to the changes to the clones in object-oriented languages. Thus, it is expected that the clones in object-oriented languages generally require less effort to be maintained compared to the clones in procedural programming languages because, higher change dispersion is a possible indicator of higher maintenance effort and cost [42].

We conclude by saying that our findings are important for taking decisions regarding cloning, clone refactoring, and clone tracking and thus, can help us in better clone maintenance. As we found Type 1 and Type 3 clones to be more unstable than Type 2 clones, Type 1 and Type 3 clones should be given a higher priority when taking clone refactoring and tracking decisions. Although, our study yields important findings regarding the comparative stability of cloned and non-cloned code, this study does not focus on the identification of the influencing factors contributing to the instability of cloned and non-cloned code. Identification of the factors that influence the instability of code might be an important area to further investigate. Future research should also be conducted to explore if or to what extent instability of code contributes to other issues in software maintenance such as bugs. Our evaluation data and the framework are now available on-line [7].

References

[1] Spearman's Rank Correlation: https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

[2] Spearman's correlation online calculator: http://www.socscistatistics.com/tests/spearman/default2.aspx

[3] Wilcoxon Signed Rank Test: https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

[4] Wilcoxon Signed Rank Test On-line Calculator: http://www.socscistatistics.com/tests/signedranks/Default2.aspx

[5] Effect Size for Wilcoxon Signed Rank Test: http://stats.stackexchange.com/questions/133077/effect-size-to-wilcoxon-signed-rank-test

[6] Effect Size: https://en.wikipedia.org/wiki/Effect_size

[7] Evaluation Data and Framework: goo.gl/1GjeZM

[8] L. Aversano, L. Cerulo, M. D. Penta,"How clones are maintained: An empirical study", Proc. *CSMR*, 2007, pp. 81–90.

[9] T. Bakota, R. Ferenc, T. Gyimothy, "Clone Smells in Software Evolution", Proc. *ICSM*, 2007, pp.24 - 33

[10] L. Barbour, F. Khomh, Y. Zou, "Late Propagation in Software Clones", Proc. *ICSM*, 2011, pp. 273-282

[11] L. Barbour, F. Khomh, Y. Zou, "An empirical study of faults in late propagation clone genealogies", *Journal of Software: Evolution and Process*, 2013, 25(11):1139 – 1165.

[12] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan, "An empirical study on inconsistent changes to code clones at release level", Proc. *WCRE*, 2009, pp. 85 – 94.

[13] D. Cai, M. Kim, "An empirical study of long-lived code clones", Proc. *FASE/ETAPS*, 2011, pp. 432 – 446.

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

[15] D. Chatterji, J. C. Carver , N. A. Kraft, "Code clones and developer behavior: results of two surveys of the clone research community", *Empirical Software Engineering*, 2016, 21(4):1476 – 1508.

[16] J. R. Cordy, and C. K. Roy, "The NiCad Clone Detector", Proc. *ICPC (Tool Demo Track)*, 2011, pp. 219 – 220.

[17] Exuberant Ctags: http://ctags.sourceforge.net/

[18] Fisher Exact Test: http://in-silico.net/tools/statistics/fisher_exact_test

[19] N. Göde, J. Harder, "Clone Stability", Proc. *CSMR*, 2011, pp. 65-74.

[20] N. Göde, R. Koschke, "Studying clone evolution using incremental clone detection", *JSME*, 2010, 25(2): 165 – 192.

[21] N. Göde, R. Koschke, "Frequency and risks of changes to clones", Proc.*ICSE*, 2011, pp.311–320.

[22] J. Harder, N. Göde, "Cloned code: stable code", *J. Softw. Evol. Process*, 25(10)(2013) 1063 – 1088.

[23] W Hordijk, M Ponisio, R Wieringa, "Harmfulness of Code Duplication - A Structured Review of the Evidence", Proc. *EASE*, 2009, pp. 88 – 97.

[24] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto, "Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software," Proc. *EVOL/IWPSE*, 2010, pp. 73–82.

[25] J. F. Islam, M. Mondal, C. K. Roy, "Bug Replication in Code Clones: An Empirical Study", Proc. *SANER*, 2016, pp. 68 – 78.

[26] S Jarzabek , Y Xu, "Are clones harmful for maintenance?", Proc. *IWSC*, 2010, pp. 73-74.

[27] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, "Do Code Clones Matter?," Proc. *ICSE*, 2009, pp. 485– 495.

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

[29] C. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software," *ESE*, 2008, 13(6): 645–692.

[30] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," Proc. *ESEC-FSE*, 2005, pp. 187–196.

[31] J. Krinke, "A study of consistent and inconsistent changes to code clones," Proc. *WCRE*, 2007, pp. 170–178.

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

[33] J. Krinke, "Is Cloned Code older than Non-Cloned Code?," Proc. *IWSC*, 2011, pp. 28 – 33.

[34] J. Li, M. D. Ernst, "CBCD: Cloned Buggy Code Detector", University of Washington Department of Computer Science and Engineering technical report UW-CSE-11-05-02, (Seattle, WA, USA), May 2, 2011. Revised October 2011.

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

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

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

[38] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, K. A. Schneider, "An Empirical Study of the Impacts of Clones in Software Maintenance", Proc. *ICPC Student Research Symposium Track*, 2011, pp. 242 – 245.

[39] M. Mondal, C. K. Roy, and K. Schneider, "An Insight into the Dispersion of Changes in Cloned and Non-cloned Code: A Genealogy Based Empirical Study", *SCP*, 2014, 95(4):445 – 468.

[40] M. Mondal, C. K. Roy, and K. A. Schneider, "An Empirical Study on Clone Stability", *ACR*, 12(3): 20–36.

[41] M. Mondal, C. K. Roy, S. Rahman, R. K. Saha, J. Krinke, K. A. Schneider, "Comparative Stability of Cloned and Non-cloned Code: An Empirical Study", Proc. *SAC*, 2012, pp. 1227 – 1234.

[42] M. Mondal, C. K. Roy, K. A. Schneider, "Dispersion of Changes in Cloned and Non-cloned Code", Proc. *IWSC*, 2012, pp. 29 – 35.

[43] M. Mondal, C. K. Roy, K. A. Schneider, "A comparative study on the intensity and harmfulness of late propagation in near-miss code clone", *Software Quality Journal*, 2016, 24(4): 883 – 915.

[44] M. Mondal, C. K. Roy, K. A. Schneider, "A Comparative Study on the Bug-Proneness of Different Types of Code Clones", Proc. *ICSME*, 2015, pp. 91 – 100.

[45] M. Mondal, C. K. Roy, K. A. Schneider, "Late Propagation in Near-Miss Clones: An Empirical Study", *ECEASST*, 63(2014): 1- 17.

[46] F. Rahman, C. Bird, P. Devanbu, "Clones: What is that Smell?", Proc. *MSR*, 2010, pp. 72 - 81.

[47] M. S. Rahman, C. K. Roy, "A Change-Type Based Empirical Study on the Stability of Cloned Code", Proc. *SCAM*, 2014, pp. 31 – 40.

[48] M. S. Rahman, A. Aryani, C. K. Roy, F. Perin, "On the Relationships between Domain-Based Coupling and Code Clones: An Exploratory Study", Proc. *ICSE NIER Track*, 2013, pp. 1265 – 1268.

[49] C. K. Roy, M. F. Zibran, R. Koschke, "The Vision of Software Clone Management: Past, Present and Future", Proc. *CSMR-18/WCRE-21 Software Evolution Week* 2014, 16 pp.

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

[51] C. K. Roy and J. R. Cordy, "A mutation / injection-based automatic framework for evaluating code clone detection tools," Proc. *Mutation*, 2009, pp. 157–166.

[52] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", *SCP*, 2009, 74(2009): 470 – 495.

[53] C.K. Roy, and J. R. Cordy, "An Empirical Evaluation of Function Clones in Open Source Software", Proc. *WCRE*, 2008, pp. 81 – 90.

[54] C. K. Roy, and J. R. Cordy, "Scenario-based Comparison of Clone Detection Techniques", Proc. *ICPC*, 2008, pp.153 – 162.

[55] R. K. Saha, C. K. Roy, and K. A. Schneider, "An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies", Proc. *ICSM*, 2011, pp. 293 – 302.

[56] 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," Proc. *SCAM*, 2010, pp. 87–96.

[57] Scorpio. http://www-sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio-e/.

[58] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, Y. Zou, "Studying the Impact of Clones on Software Defects", Proc. *WCRE*, 2010, pp.13 – 21.

[59] Simian-similarity analyser. http://www.redhillconsulting.com.au/products/simian/

[60] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta, "An empirical study on the maintenance of source code clones," *ESE*, 2009, 15(1):1–34.

[61] T. Wang, M. Harman, Y. Jia, J. Krinke, "Searching for Better Configurations: A Rigorous Approach to Clone Evaluation", Proc. *ESEC/SIGSOFT FSE*, 2013, pp. 455–465.

[62] X. Wang , Y. Dang , L. Zhang , D. Zhang , E. Lan , H. Mei, "Can I clone this piece of code here?", Proc. *ASE*, 2012, pp. 170 – 179.

[63] N. Göde and R. Koschke., "Incremental clone detection", Proc.*CSMR*, 2009, pp. 219–228.

[64] Y. Higo and S. Kusumoto., "Significant and Scalable Code Clone Detection with Program Dependency Graph.", Proc. *WCRE*, 2009, pp. 315–316.