Dispersion of Changes in Cloned and Non-cloned Code

Manishankar Mondal Chanchal K. Roy Kevin A. Schneider
University of Saskatchewan, Canada
{*mshankar.mondal, croy@cs.usask.ca, kevin.schneider*}*@usask.ca*

***Abstract*—Currently, the impacts of clones in software maintenance activities are being investigated by different researchers in different ways. Comparative stability analysis of cloned and non-cloned regions of a subject system is a well-known way of measuring the impacts where the hypothesis is that, the more a region is stable the less it is harmful for maintenance. Each of the existing stability measurement methods lacks to address one important characteristic, *dispersion*, of the changes happening in the cloned and non-cloned regions of software systems. Change *dispersion* of a particular region quantifies the extent to which the changes are scattered over that region. The intuition is that, more dispersed changes require more efforts to be spent in the maintenance phase.**

Measurement of *Dispersion* requires the extraction of method genealogies. In this paper, we have measured the dispersions of changes in cloned and non-cloned regions of several subject systems using a concurrent and robust framework for method genealogy extraction. We implemented the framework on Actor Architecture platform which facilitates coarse grained parallellism with asynchronous message passing capabilities. Our experimental results with 12 open-source subject systems written in three different programming languages (Java, C and C#) using two clone detection tools suggest that, the changes in cloned regions are more dispersed than the changes in non-cloned regions. Also, Type-3 clones exhibit more dispersion as compared to the Type-1 and Type-2 clones. The subject systems written in Java and C show higher dispersions as well as increased maintenance efforts as compared to the subject systems written in C#.

***Keywords*-Dispersion; Changeability; Code Stability; Modification Frequency; Average Last Change Date; Actor Architecture; Concurrent Framework;**

I. Introduction

Reuse of code fragments with or without modifications by copying and pasting from one location to another is very common in software development. This results in the existence of same or similar code blocks in different components of a software system. Code fragments that are exactly same or very similar to each other are known as clones. The impacts of clones are of great concern from the maintenance point of view. Researchers have tried to find out the impacts of clones indirectly by determining the comparative stabilities of cloned and non-cloned code from different perspectives. The intuition is that, if cloned code is more stable (gets less frequent changes) than non-cloned code in general, cloned code does not increase maintenance effort and is not harmful for the maintenance phase as well.

There are a number of existing methods for calculating stabilities of cloned and non-cloned code of a subject system. Most of these [4], [5], [10], [13] calculate stability in terms of instability or changeability. Only one method [11] calculate stability in terms of age of LOC. Instability has been measured in two ways (i) by the ratio of the total number of lines added, deleted and modified in a code region to the total number of lines of the code region and (ii) by determining the modification frequency of a code region where the modification frequency [5] considers the number of occurrences of addition, deletion and modification. Each occurrence can consist of several consecutive lines. The remaining approach [11] calculates the average last change dates of cloned and non-cloned code regions using the blame command of SVN.

Each of these existing approaches lacks some important information. These are about whether the changes are taking place to the same regions repeatedly or to different regions during the evolution of a software system. This information is very important for analyzing code stability as well as its impact. Repeated changes to the same code region (e.g. method) are more manageable as compared to the scattered changes at different regions. If a change takes place to a method for the first time, programmers need to spend a considerable amount of time to understand the method context and to determine the possible impacts of the changes to other related code regions (or methods). This might be necessary that relevant changes need to be propagated to other similar code fragments (clones) to maintain consistency. Even a very small change might have a great impact to the whole software system (According to a recent study [6], every second unintentional inconsistent change to a clone leads to a fault). However, further changes to the same method do not possibly require much effort because the impact analysis of changes to this method has already been done. Thus, several changes to different code regions generally are more difficult to tackle than those changes in the same code region. We can thus say that, if during the evolution of a software system, C changes take place to n_1 different regions of cloned code and the same number of changes (C) take place to n_2 different regions of non-cloned code for the same number of consecutive revisions where $n_1 > n_2$, then, for this software system non-cloned code is more stable than cloned code because modifications in the cloned code require more effort to be managed than those of its non-cloned counterpart.

To incorporate this information in the stability measurement process we have introduced *dispersion*, a new stability measurement metric, which we have defined by

the percentage of unique methods affected by changes in cloned or non-cloned regions. We have calculated dispersion using method level granularity. For determining whether the changes are taking place to the same or different locations (methods) during the evolution we have extracted method genealogies from all the revisions of a subject system.

During the evolution of a software system a particular method might be created in a particular revision and can remain alive in multiple consecutive revisions. Each of these revisions has separate instances of this method. Method genealogy identifies all of these instances as belonging to the same method. Extraction of method genealogies is a complex and time consuming task which can be divided into multiple smaller independent tasks that can be executed in parallel to reduce the time complexity of sequential calculation. In this paper, we have also proposed and implemented a parallel and distributed framework for method genealogy extraction and used the framework for calculating change dispersion.

Our experimental results on 12 subject systems covering three different programming languages considering all three major types (Type-1, Type-2,Type-3) of clones using two clone detection tools (CCFinderX [3] and NiCad [16]) indicate that, dispersion of changes in the cloned code is higher than the dispersion of changes in the non-cloned code. In other words, the percentage of methods affected by changes in cloned code is greater than the percentage of methods affected by the changes in non-cloned code. Thus, cloned code is possibly more vulnerable than the non-cloned code to the maintenance phase. We have also found that, Type-3 clones exhibit higher dispersion as compared to the Type-1 and Type-2 clones. The subject systems written in Java and C show higher dispersions in the maintenance phase as compared to the C# systems.

The rest of the paper is organized as follows: Section II outlines the relevant research, Section III elaborates dispersion, the concurrent framework for method genealogy extracton is described in Section IV and Section V extends the concurrent framework for calculating dispersion. Section VI contains the experimental setup. The experimental result is presented in Section VII. Section VIII describes possible threats to validity and Section IX contains concluding remarks and future works.

II. Related Work

Over the last several years, the impact of clones has been an area of focus for software engineering research resulting in a significant number of studies and empirical evidence. Kim et al. [8] proposed a model of clone genealogy. Their study with the revisions of two medium sized Java systems showed that refactoring clones may not always improve software quality. They also argued that aggressive and immediate refactoring of short-lived clones is not required and that such clones might not be harmful. Saha et al. [17] extended their work by extracting and evaluating code clone genealogies at the release level of 17 open source systems involving four different languages. Their study reports similar findings to Kim et al. and concludes that most of the clones do not require any refactoring effort.

Kapser and Godfrey [7] strongly argued against the conventional belief of harmfulness of clones. In their study they identified different patterns of cloning and showed that about 71% of the cloned code has a kind of positive impact in software maintenance. They concluded that cloning can be an effective way of reusing stable and mature features.

Lozano and Wermelinger [14] developed a prototype tool to track the frequency of changes of cloned and non-cloned code with method level granularity. On the basis of their study on four open source systems they concluded that the existence of cloned code within a method significantly increases the required effort to change the method. In a recent study [13] they further analyzed clone imprints over time and observed that cloned methods remain cloned most of their life time and cloning introduces a higher density of modifications in the maintenance phase.

Juergens et al. [6] studied the impact of clones on large scale commercial systems and suggested that inconsistent changes occurs frequently with cloned code and nearly every second unintentional inconsistent change to a clone leads to a fault. Aversano et al. [2] on the other hand, carried out an empirical study that combines the clone detection and co-change analysis to investigate how clones are maintained during evolution or bug fixing. Their case study on two subject systems confirmed that most of the clones are consistently maintained. Thummalapenta et al. [19] in another empirical study on four subject systems concluded that most of the clones are changed consistently and other inconsistently changed fragments evolve independently.

In a recent study [4] Göde and Harder replicated and extended Krinke's study [10] using an incremental clone detection technique 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 while this scenario reverses with respect to deletions.

Hotta et al. [5] studied the impact of clones by measuring the modification frequencies of cloned and noncloned code of several subject systems. Their study using different clone detection tools suggests that the presence of clones does not introduce extra difficulties to the maintenance phase.

Krinke [9] measured how consistently the code clones are changed during maintenance using *Simian* [18] and *diff* 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 [10]. In his most recent study [11] he calculated the average last change dates of the cloned and non-cloned code and observed that cloned code is more stable than non-cloned code.

None of the existing studies have measured the dispersion of changes. But, without measuring dispersion we cannot accurately measure the impact of a particular region (cloned or non-cloned). We have introduced and measured dispersion in this paper. Our experimental results suggest that, the changes in the cloned regions are more dispersed than the changes in the non-cloned regions of a subject system.

III. Calculation of Dispersion

We consider method level granularity for measuring dispersion. A method is defined as a cloned method when it contains some cloned lines in it. According to our consideration 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). For calculating the dispersion of cloned code, we consider the changes in the cloned portions of the cloned (fully or partially) methods. Partially cloned methods have also been considered while calculating the dispersion of non-cloned code because, changes might occur in the non-cloned portions of the partially cloned methods. Also, 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 have been considered in calculating the dispersions of both cloned and non-cloned code.

Suppose, we have a subject system of R revisions. At first, we find the methods and their boundaries in each revision. Then, we extract the method genealogies from these revisions and determine the *unique methods*. A *unique method* is a method which gets created in a particular revision and lives in several consecutive revisions with or without changes. Then, we apply a clone detection tool in these revisions to determine the clone blocks. By determining which clone block is contained in which method, we calculate the counts of unique cloned and non-cloned methods. We also determine the changes between consecutive revisions and reflect these changes to the cloned and non-cloned portions of the methods.

Suppose, for a subject system, the counts of unique cloned and unique non-cloned methods are C and N respectively. C_c is the number of unique cloned methods which got some changes in their cloned portions during the evolution. The number of changes received by C_c unique cloned methods is generally greater than C_c. Also, N_c is the number of unique non-cloned methods that received some changes in their non-cloned portions. If we denote the change dispersions of cloned code and non-cloned code by CD_c and CD_n respectively, they can be expressed by the following equations.

$$CD_c = \frac{C_c \times 100}{C} \tag{1}$$

$$CD_n = \frac{N_c \times 100}{N} \tag{2}$$

IV. Concurrent framework for method genealogy extraction

The huge task of method genealogy extraction can be divided into multiple smaller tasks which can be executed by different processes in parallel on different processors of the same machine or on different machines connected through an interconnection network. The results of these tasks can be combined to achieve the final result. We need to emphasize on the following objectives during the division of tasks.

(1) The tasks should not be too small so that, the processes can spend more time on task completion rather than inter-process communication.

(2) The processes should synchronize among themselves to ensure the consistency of execution.

(3) Task distribution should ensure load balancing.

Focusing on these objectives, we have defined our genealogy extraction model as a 'Manager-Workers' paradigm where there is a single manager who manages or coordinates the tasks of several workers. At the very begining of execution, manager divides the whole range of revisions into a number of sub-ranges of equal length. Each sub-range contains multiple consecutive revisions and the count of sub-ranges is equal to the number of workers. The manager then assigns each of the sub-ranges to a particular worker. Each worker is responsible for the extraction and mapping of the methods of the revisions it has been assigned to. To get the final method mapping for the whole range of revisions, the workers need to synchronize among themselves. The synchronization process is described below with an example.

Suppose, two workers worker1 and worker2 are responsible for revisions with ranges 1 to 10 and 11 to 20 respectively. Each worker will have to complete the extraction and mapping of methods of its respective revisions. Each worker is disciplined in such a way that, it at first extracts the methods of i^{th} revision in its range, stores the methods with associated information into a file and then maps the methods remaining in the files resulted from i^{th} and $(i-1)^{th}$ revisions. Then, the worker proceeds with the $(i+1)^{th}$ revision. With this discipline, worker2 will be able to extract the methods of 11^{th} revision but will not be able to complete the mapping between 10^{th} and 11^{th} revisions. Because, worker2 does not know whether the methods from 10^{th} revision have been extracted and stored by worker1. Worker1 and worker2 are executing in parallel by starting their execution at arround the same time and are processing their respective range of revisions begining with the very first revisions of their ranges. With proper load balancing, it is likely that, worker1 and worker2 will be processing respectively the 10^{th} and 20^{th} revisions at around the same time. In this case, worker1 needs to send a message to worker2 after it has extracted and stored the methods of 10^{th} revision. Worker2, in this situation, extracts methods of revisions 11 to 20 and performs mapping on the revisions 12

to 20 and then waits for the message from worker1. After getting the message, worker2 performs mapping between revesions 10 and 11.

V. Calculating Dispersion using Concurrent Framework

The activities of the manager and workers can be extended for calculating dispersion. In this case, the manager not only distributes the revisions to the workers but also collects the values of four counters (C, N, C_c, N_c) from the workers. By calculating the final values of these four counters, manager calculates the dispersions of cloned and non-cloned code.

In this case, after extracting the genealogies, each worker should send message to all other workers to inform that, it has completed extracting genealogies for the revisions of its range. After getting messages from all other workers, a worker begins examining the genealogies of unique methods belonging to the revisions of its range. As the workers examine the method genealogies of their ranges, they update their respective set of counters (C, N, C_c, N_c). At the end of examination, they send these counters to the manager.

The following sections describe how we have downloaded the subject systems and how we have calculated the dispersions of cloned and non-cloned code.

A. Extraction of repositories

All of the subject systems (listed in Table II) on which we have applied our method to calculate dispersion have been downloaded from open source SVN repositories. For a subject system, we extracted only those revisions which were created because of some source code modification (addition, deletion or change). 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 two preprocessings on the source files of each target revision of a subject system before clone detection. These are - **(i)** deletion of lines containing only a single brace ('{' or '}') and appending the brace at the end of the previous line and **(ii)** removal of comments and blank lines.

C. Method detection and extraction

For detecting the methods we applied CTAGS on the source files of a revision. For each method we collected - (i) file name, (ii) class name (Java and C# systems), (iii) package name (Java), (iv) method name, (v) signature, (vi) starting line number and (vii) ending line number. We also assigned a unique id to each method. However, the id of a method of one revision can be the same as that of a method of another revision. This does not introduce conflicts because separate file is generated for each revision.

D. Clone detection

We have used CCFinderX [3] and NiCad [16] for detecting clones in our experiment. CCFinderX is a token based clone detection tool that currently detects block clones of Type-1 and Type-2. Also, NiCad is a recently introduced clone detection tool that can detect three types of clones (Type-1, Type-2, Type-3) with high precision and recall [15] considering both block level and method level granularities.

We applied these two 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 begining 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.

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 (i) addition, (ii) deletion and (iii) modification with corresponding line numbers. We mapped these changes to methods using line information. So, for each method we gathered two more information - the count of lines changed in cloned portions and the count of changed lines in non-cloned portions.

F. Storage of methods

At this stage, we have got all necessary pieces of information of all methods belonging to a particular revision. We store these methods in an xml file with individual entry for each method. For each revision we generated separate xml file containing the methods of the corresponding revision. A file name is constructed by appending the revision number at its end so that, we can generate it when necessary (for getting previously stored methods of the unchanged files of a former revision and for calculating dispersion).

G. Method mapping

For mapping methods between two consecutive revisions we followed the origin analysis technique proposed by Lozano and Wermelinger [14]. This technique uses a combination of location and signature similarities to determine which method of revision i corresponds to which method of revision i+1. Some methods of revision i might get deleted in revision i+1 and also some new methods might be created in the (i+1)-th revision. We stored the mapping information for each two consecutive revisions in a separate file. Method mapping was accomplished using method ids. The file names contain the revision numbers in a disciplined way so that we can generate them when necessary.

H. Calculation of dispersion

For calculating dispersion we examine the genealogy of each *unique method* and during this examination we update four counters - **(i)** the count of unique cloned methods (C) **(ii)** the count of unique non-cloned methods (N) **(iii)** the count of unique cloned methods which have got some changes in their cloned portions (C_c) and **(iv)** the count of unique non-cloned methods which have got some changes in their non-cloned portions (N_c).

For examining the genealogy of a particular *unique method* we at first determine the revision in which it was created. Begining with this revision we examine the instances of the method in all consecutive revisions where it was alive. While examining the genealogy of a particular method we updated the counters in the following way.

- If any of these instances contain a cloned portion, we increment the counter C by 1.
- If one or more of these instances contain a non-cloned portion, we increment N by 1.
- If one or more of these instances get some changes in their cloned portions, we increment C_c by 1.
- If one or more of these instances get some changes in their non-cloned portions, we increment N_c by 1.

This is obvious that, the genealogy of a particular unique method can increment each of the counters by at most 1.

In the parallel and distributed environment, each of the workers is assigned a particular range of revisions. A particular worker examines the genealogies of only those methods which have been created in the revisions they are responsible for. Each worker has its respective set of four counters which are updated by this worker only. After examining all the unique methods of all the revisions in a range, the associated worker sends its copy of four counters to the manager. The manager gets all the instances of each of the counters, adds the respective instances to get the final values of these counters and calculates the dispersions (CD_c, CD_n) using the equations Eq. 1 and Eq. 2 respectively.

I. Message passing technique

To provide message passing facility to the manager and the workers, we implemented them in Java using Actor Architecture platform [1]. The Actor Architecture platform provides a class named 'Actor' which is extended by each of the workers and the manager. Before activating the manager and workers in a machine we at first need to run the Actor Architecture platform on that machine. If the workers are distributed in more than one machine, each of the machines needs to run the platform.

VI. Experimental Setup

A. Setup for CCFinderX

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 I: NiCad Settings

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

Table II: Subject Systems

	Systems	Domains	LOC	Revisions
Java	DNSJava	DNS protocol	23,373	1635
	Ant-Contrib	Web Server	12,621	176
	Carol	Game	25,092	1699
	jabref	Project Management	79,853	32
C	Ctags	Code Def. Generator	33,270	774
	Camellia	Multimedia	85,015	55
	QMail Admin	Mail Management	4,054	317
	Gnumakeuniproc	Project Building	83,269	110
C#	GreenShot	Multimedia	37,628	999
	ImgSeqScan	Multimedia	12,393	73
	Capital Resource	Database Management	75,434	122
	MonoOSC	Formats and Protocols	18,991	355

B. Setup for NiCad

Using NiCad 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 I 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. For all the settings in Table I NiCad was shown to have high precision and recall [15]. Before using the NiCad outputs of Type-2 and Type-3 cases, we processed them in the following ways.

(1) Every Type-2 clone class that exactly matched any Type-1 clone class was excluded from Type-2 outputs.

(2) Every Type-3 clone class that exactly matched any Type-1 or Type-2 clone class was excluded from Type-3 outputs.

C. Subject Systems

Table II lists the subject systems that were included in our study along with their associated attributes. We selected this set of subject systems for a number of reasons. The subject systems are diverse, differing in size, spanning 10 different application domains, and covering three different programming languages.

D. Concurrent framework

We have implemented the concurrent framework for calculating dispersion in Java programming language using the Actor Architecture platform [1]. We executed our framework on multiple machines connected with an interconnection network. The concurrent framework was significantly faster than the sequential program which we implemented initially for genealogy extraction.

Table III: Comparative dispersions of Cloned and Non-cloned Methods

Lang	Systems	Type 1			Type 2			Type 3			CCFinder		
		CD_c	CD_n	Rem	CD_c	CD_n	Rem	CD_c	CD_n	Rem	CD_c	CD_n	Rem
Java	DNSJava	24.53	5.91	⊖	15.17	7.82	⊖	18.16	7.55	⊖	18	7	⊖
	Ant-Contrib	17.64	1.63	⊖	2.22	1.95	⊖	5	1.97	⊖	5	0	⊖
	Carol	5.87	19.50	⊕	9.35	19.33	⊕	18.92	19.72	⊗	23	7	⊖
	jabref	11.23	20.43	⊕	8.95	21.78	⊕	13.05	18.44	⊕	31	8	⊖
C	Ctags	0	10.04	⊕	20	9.66	⊖	14.53	9.78	⊖	21	10	⊖
	Camellia	0	9.85	⊕	12.5	9.55	⊖	35	8.76	⊖	30	9	⊖
	QMail Admin	50	7.29	⊖	42.85	8.03	⊖	60	8.15	⊖	50	12	⊖
	Gnumakeuniproc	12.5	0.42	⊖	0	0.50	⊕	1.38	0.51	⊖	1.26	2.73	⊕
C#	GreenShot	8.88	29.32	⊕	22.96	29.49	⊕	30.37	30.47	⊗	14	5	⊖
	ImgSeqScan	0.0	3.76	⊕	0.0	3.73	⊕	0.0	3.72	⊕	25	0.55	⊖
	Capital Resource	0.0	4.92	⊕	0.0	4.79	⊕	3.95	4.47	⊕	3.95	4.58	⊕
	MonoOSC	3.17	10.48	⊕	5.26	10.42	⊕	39.13	10.03	⊖	39.44	12	⊖

CD_c= *Dispersion of Changes in Cloned Methods* CD_n= *Dispersion of Changes in Non-cloned Methods*

⊕ = $CD_c < CD_n$ ⊖ = $CD_c > CD_n$

⊗ = *Difference between the dispersions of cloned and non-cloned methods is not significant.*

VII. Experimental Results and Discussion

The magnitudes of dispersions of cloned and non-cloned code for each of the subject systems with associated remarks are shown in the Table III which contains 48 (12 subject systems × 4 cases) decision points. Each point consists of a particular subject system and a particular case (Type1, Type2, Typ3 or CCFinder) and contains corresponding dispersions (CD_c, CD_n) and a remark (Rem). The decision points are categorized into three categories which are explained below.

Category 1. For these points (indicated by ⊕), the changes in cloned code were less dispersed than the changes in non-cloned code ($CD_c < CD_n$).

Category 2. The decision points marked with ⊖ signs indicate that changes in non-cloned code were less dispersed than those of cloned code ($CD_n < CD_c$).

Category 3. For these points (marked with ⊗), we have not got significant differences between the corresponding dispersions of cloned and non-cloned code.

To determine whether the difference between the dispersions of cloned and non-cloned code of a subject system for a particular case is significant we calculated an $Eligibility\ Value$ on the observed dispersions according to the following equation.

$$EligibilityValue = \frac{(HCD - LCD) * 100}{LCD} \quad (3)$$

Here, HCD stands for Higher Change Dispersion where LCD is elaborated as Lower Change Dispersion. An eligibility value of at least 10 is treated as a significant one.

If this $EligibilityValue$ is greater than the threshold value, the subject system will be counted as belonging to $Category1$ or $Category2$ (if $CD_c < CD_n$ then $Category1$ otherwise $Category2$). We have selected the calculation procedure and the threshold magnitude of $ElibigilityValue$ in such a way that, they will force a subject system having larger but very near dispersions (such as 41 and 40. $EligibilityValue$ = (41-40)*100/40 = 2.5) to be selected in $Category3$ while a subject system with smaller but near dispersions (such as 3 and 4. $EligibilityValue$ = 33.33) to be selected in $Category1$ or $Category2$ which is expected.

Overall analysis: Among 48 decision points of Table III, 46 points fall in category 1 or category 2. We call them *significant decision points* because the differences between the dispersions for these points are significant according to the *Eligibility Value* of equation Eq. 3. We have ignored the remaining 2 decision points marked with (⊗).

According to 44.44% of the significant points, dispersion of changes in cloned code is less than the dispersion of changes in non-cloned code. The opposite is true for the remaining 55.56% points. Though the difference between the percentages is not much significant, it indicates that, *the 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*.

Language centric analysis: From the graph in Fig. 1 we see that, in case of Java, 66.67% of the significant decision points suggest higher dispersion of changes in the cloned code, whereas it is 75% for C and 26.67% for C#. Thus, as regards to dispersion, *both Java and C are possibly more vulnerable for the maintenance phase with C being the most threatened one. The vulnerability imposed by the clones of C# systems is much lower than the correspondig non-cloned code and thus can be ignored.*

Type centric analysis: According to the type centric statistics of the graph in Fig. 2, *each of the clone types of Java and C poses some vulnerability to the maintenance phase with Type-3 being the most vulnerable one for each of these two languages*. Our results also show that, *no decision points belonging to Type 1 and Type 2 cases of C# exhibit*

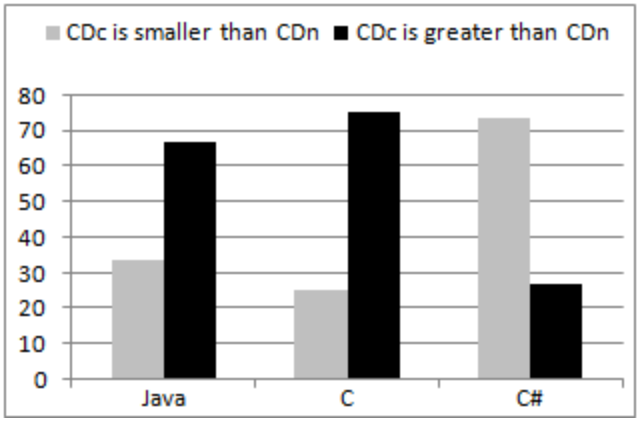

Figure 1: Programming language centric statistics

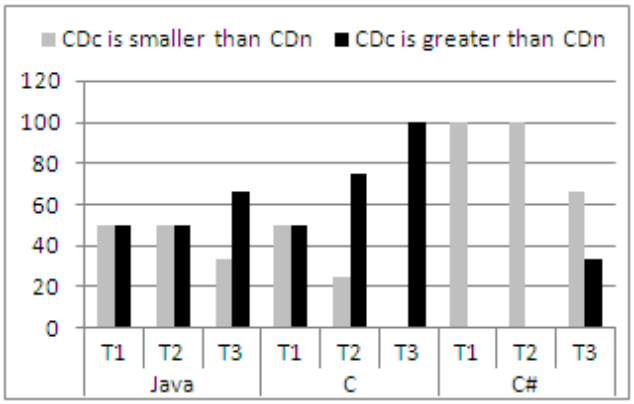

Figure 2: Type centric statistics for each prog. language

higher dispersion of cloned code. Thus, we possibly do not need to pay much attention for these two types of clones of C#. However, Type 3 clones of this language should be taken care of.

VIII. Threats to Validity

We calculated and analyzed the dispersions of changes of cloned and non-cloned code for only 12 subject systems which are no way sufficient for taking any general decision about different types of clones and programming languages. Also, some other important factors such as programmer expertise, application domain, programmer‘s knowledge about appilcation domain were not considered in our experiment. But, our selection of subject systems considering ten application domains, three programming languages, diversified sizes and revisions have considerably minimized these lackings, and thus we believe that our findings are significant.

IX. Conclusion

In this paper, we have introduced a new metric *dispersion* (of changes) which measures how much scattered the changes in a particular code region (cloned or non-cloned) are. Calculation of dispersion requires the extraction of method genealogies. We have also proposed a parallel and distributed framework for method genealogy extraction and extended this framework for calculating dispersion. We implemented this extended framework on Actor Architecture platform. Our concurrent framework will help the future software engineering research (that involve the processing of multiple revisions) by reducing a significant amount of processing time. The introduced metric will assist in fine grained calculation of the impacts of cloned and non-cloned code in the maintenance phase. Our experimental result on 12 open source subject systems written in three different programming languages suggests that, the changes in cloned code are more dispersed than the changes in non-cloned code. So, according to the dispersion of changes, cloned code requires more maintenance effort than non-cloned code. Also, Type-3 clones show more dispersed changes as compared to the other two clone types (Type-1 and Type-2). Moreover, the subject systems written in Java and C show higher dispersions than the subject systems written in C#. We are planning to investigate the dispersions of cloned and non-cloned code for different application domains to determine which type of applications generally exhibit higher change dispersions.

References

[1] Actor Architecture platform. http://www.docstoc.com/docs/5693312/Actor-Architecture.
[2] L. Aversano, L. Cerulo, M. D. Penta, “How clones are maintained: An empirical study”, in *Proc. CSMR*, 2007, pp. 81-90.
[3] CCFinderX. http://www.ccfinder.net/ccfinderxos.html
[4] N. Göde, J. Harder, “Clone Stability”, in *Proc. CSMR*, 2011, pp. 65-74.
[5] 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”, in *Proc. EVOL/IWPSE*, 2010, pp. 73-82
[6] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, “Do Code Clones Matter?”, in *Proc. ICSE*, 2009, pp. 485-495.
[7] C. Kapser and M. W. Godfrey, ““Cloning considered harmful” considered harmful: patterns of cloning in software”, *Emp. Soft. Eng.* 13(6), 2008, pp. 645-692.
[8] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, “An empirical study of code clone genealogies”, in *Proc. ESEC-FSE*, 2005, pp. 187-196.
[9] J. Krinke, “A study of consistent and inconsistent changes to code clones”, in *Proc. WCRE*, 2007, pp. 170-178.
[10] J. Krinke, “Is cloned code more stable than non-cloned code?”, in *Proc. SCAM*, 2008, pp. 57-66.
[11] J. Krinke, “Is Cloned Code older than Non-Cloned Code?”, in *Proc. IWSC*, 2011, pp. 28-33.
[12] A. Lozano, M. Wermelinger, B. Nuseibeh, “Evaluating the Harmfulness of Cloning: A Change Based Experiment”, in *Proc. MSR*, 2007, pp. 18-21.
[13] A. Lozano and M. Wermelinger, “Tracking clones’ imprint”, in *Proc. IWSC*, 2010, pp. 65-72.
[14] A. Lozano, and M. Wermelinger, “Assessing the effect of clones on changeability”, in *Proc. ICSM*, 2008, pp. 227-236.
[15] C. K. Roy and J. R. Cordy, “A mutation / injection-based automatic framework for evaluating code clone detection tools,” in *Proc. ICST Mutation*, 2009, pp. 157–166.
[16] C.K. Roy and J.R. Cordy, “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization,” in *Proc. ICPC*, 2008, pp. 172–181.
[17] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, “Evaluating code clone genealogies at release level: An empirical study”, in *Proc. SCAM*, 2010, pp. 87-96.
[18] “Simian - Similarity Analyser”. http://www.harukizaemon.com/simian/index.html.
[19] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta, “An empirical study on the maintenance of source code clones”, in *Emp. Soft. Engg.*, 15(1), 2009, pp. 1-34.