Insight into a Method Co-change Pattern to Identify Highly Coupled Methods: An Empirical Study

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

***Abstract*—In this paper, we describe an empirical study of a unique method co-change pattern that has the potential to pinpoint design deficiency in a software system. We automatically identify this pattern by inspecting the method co-change history using reasonable constraints on method association rules. We also investigate the effect of code clones on the method co-changes identified according to the pattern, because there is a common intuition that clone fragments from the same clone class often require corresponding changes to ensure they remain consistent with each other.**

According to our in-depth investigation on hundreds of revisions of seven open-source software systems considering three types of clones (Type 1, Type 2, Type 3), our identified pattern helps us detect methods that are logically coupled with multiple other methods and that exhibit a significantly higher modification frequency than other methods. We call the methods detected by the pattern *MMCG*s (Methods appearing in Multiple Commit Groups) considering the pattern semantic. *MMCG*s can be considered as the candidates for restructuring in order to minimize coupling as well as to reduce the change-proneness of a software system. According to our observation, code clones have a significant effect on method co-changes as well as on *MMCG*s. We believe that clone refactoring can help us minimize evolutionary coupling among methods.

***Index Terms*—Association Rules, Evolutionary Coupling, Life Span, Modification Occurrence Rate, Method Co-change Pattern, Method Genealogy.**

I. Introduction

Software maintenance is one of the most important phases of the software development life cycle. Changes to a software system during maintenance are sometimes critical. A particular change to a program entity (e.g., files, classes, methods) without proper awareness of its dependencies (or coupling) might cause temporarily hidden inconsistencies in other related entities. Identification and prediction of co-changing program entities by analyzing the software evolution history is a well known way of increasing programmer awareness about entity coupling. The underlying idea is that if two or more entities change together (i.e., co-change) frequently during software evolution, then it is very likely that there is a dependency, that means logical coupling, among these entities. Co-changeability of program entities has also been termed 'evolutionary coupling' [21] in the literature.

There are a number of studies [5]–[8], [11], [12], [21], [25]–[29] and tools [3], [4], [10] that focused on the identification, visualization, and prediction of co-changeable program entities. Co-change analysis can help us to detect entity-couplings which might not be detected by program analysis [28].

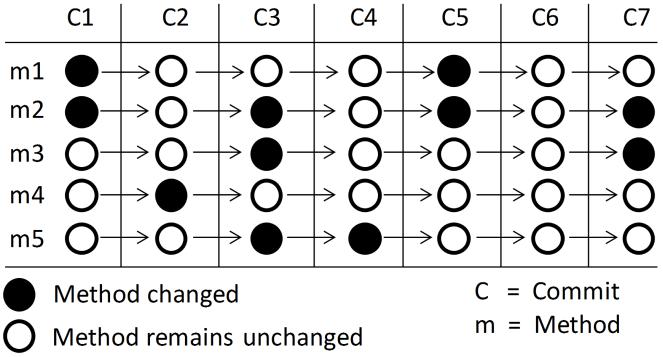

Fig. 1. An example of our investigated method co-change pattern

In this research work, we investigate whether we can detect highly coupled program entities by analyzing the entity co-change history. Detection of highly coupled entities is important, because the existence of such entities might be indicative of design deficiencies in the software system. Generally, lower couplings or dependencies among program entities are always desirable [20]. When a software system has a high degree of entity-coupling, changes in one entity might affect other related entities (ripple effect), and therefore, re-usability of the code is reduced[1]. Thus, highly coupled entities are candidates for restructuring in order to minimize coupling. We conducted this research considering method level granularity. For the purpose of our investigation we detected and analyzed a unique method co-change pattern that has the potential to identify highly coupled methods. We define the pattern in the following way.

Pattern Definition: Suppose $m1$, $m2$, and $m3$ are three methods of a particular software system. During system evolution, these methods changed, such that:

- whenever $m1$ changed $m2$ had a corresponding change
- whenever $m3$ changed $m2$ had a corresponding change
- $m1$ and $m3$ sometimes (or never) co-changed.

In Fig. I, the methods: $m1$, $m2$, and $m3$ have changed following such a pattern. We see that $m1$ changed in two commit operations: $C1$ and $C5$. The method $m2$ changed in each of these two commits too. Also, in each of the commits $C3$ and $C7$, both $m3$ and $m2$ changed together. Thus, $m1$ and $m3$ were changed with $m2$ making two separate commit groups (($m1$, $m2$) and ($m2$, $m3$)) and $m2$ is common in these

[1]Software Coupling. http://www.avionyx.com/publications/e-newsletter/issue-3/126-demystifying-software-coupling-in-embedded-systems.html

two groups. This is why we denote $m2$ as a *Method appearing in Multiple Commit Groups* (MMCG). From Fig. I we see that such a strict pattern cannot be obtained by considering any of the other two methods $m4$ and $m5$.

After manually investigating a large number of such patterns from software evolution histories, we found that in most cases, *MMCG*s (e.g. $m2$) are logically coupled with the methods in the commit groups (e.g., $m1$ and $m3$). However, although we described the above pattern using only three methods and two commit groups, we observed many such patterns where an *MMCG* appears in more than two commit groups and are logically coupled with more than two other methods. Thus, such a co-change pattern can help us in detecting highly coupled *MMCG*s.

In this paper we devised and implemented a technique for extracting *MMCG*s detecting the method co-change pattern and ranking *MMCG*s according to the number of other methods with which they are logically coupled. Our implementation automatically inspects the method co-change history with reasonable constraints from hundreds of commits of a software system for identifying such patterns.

We also investigated the effect of code clones [9] on method co-changes that follow the above pattern. Clones are the same or similar code fragments scattered in the code base. Clone fragments that are similar to one another form a clone class. The common intuition is that clones in the same clone class might require corresponding changes to ensure consistency among them and thus, clones belonging to the same class have a tendency to co-change.

We perform a rigorous manual analysis on the detected *MMCG*s. Our in-depth investigation on hundreds of revisions of seven open source subject systems written in three different programming languages (C, Java, C#), indicates that: **(1)** Our proposed approach, implemented for detecting the pattern, can identify *MMCG*s that are logically coupled with multiple other methods, **(2)** Instability or change-proneness of the *MMCG*s is significantly higher compared to the other methods, **(3)** *MMCG*s live significantly longer than the other methods, and **(4)** The proportion of *MMCG*s is positively correlated to the instability of the entire software system (that is, the higher the instability of a software system the higher the proportion of *MMCG*s in that system).

From our results we realize that: **(1)** a possible cause for the higher modification probability of *MMCG*s is that they are logically coupled with multiple other methods; and, **(2)** as *MMCG*s appear to be the most change-prone methods as well as the longest lived methods in a software system, it is very important for a new maintenance programmer to be aware and have an understanding of these methods and their relationships. Thus, the defined pattern, and the *MMCG*s are important from the perspective of program comprehension.

Our investigation regarding the effect of clones suggest that a considerable amount of method co-changes can occur because of the consistency ensuring changes to the clones of the same clone class. Sometimes a significant portion of the *MMCG*s can exhibit evolutionary coupling because of such consistency ensuring changes in clones. We believe that refactoring of clones can help to minimize evolutionary coupling among methods.

The remainder of the paper has been organized as follows: Section II describes how our study is significant compared to the related studies. Section III describes the detection procedure of the pattern and *MMCG*s. Section IV describes our experimental setup and steps used in our experiment. Section V provides and discusses the experimental results. Section VI describes some possible threats to validity and we conclude the paper by mentioning our future work in Section VII.

II. Significance of Our Study

To the best of our knowledge, our study is the first one regarding the identification of highly coupled program entities (methods in our study) in a software system through the analysis of the entity co-change history.

The primary focus of the existing studies [5]–[7], [11], [12], [15], [21], [26]–[29] was on the accuracy of detecting and predicting co-changeable program entities. Most of these studies were conducted on file level co-changeability [5]–[7], [11], [12], [15], [21], [26]. Only a few [27]–[29] investigated the co-changeability of finer grained entities such as classes, method, variables, and lines. None of these studies investigated how to detect highly coupled entities analyzing the entity co-change history and how to minimize their coupling.

A number of studies [2], [11], [23], [24] were conducted on the identification of possible structural weaknesses analyzing the co-evolution of files [2], [23], [24] or system modules [11]. The underlying idea is that if two or more files from different folders co-change frequently, this might be an indication of structural weakness. We believe that existence of highly coupled methods is another indicator of structural weakness.

The existing co-change visualization tools: *Evolution Radar* [10], *CCVisu* [3], *Evolution Storyboard* [4], interactive visualization [23] can clearly indicate file level co-changes. These tools cannot detect method level co-changes. *Rose* tool [28], implemented as a plug-in, can detect which other entities (variables, methods, classes, files) we might need to change while changing a particular entity. Although this tool considers the co-changes among finer grained entities, it was not designed for detecting highly coupled methods.

In this paper, we advance the state of the art by not only considering the co-change of a finer grained entity, method, but also proposing, detecting, and investigating a unique method co-change pattern that has the potential to pinpoint highly coupled methods (possible places of restructuring to minimize coupling) in a software system.

III. Detection of the Pattern and MMCGs

We detect the method co-change pattern as well as *MMCG*s by mining and inferring association rules. Association rules have already been used to determine co-changing program entities [28]. For detecting the particular co-change pattern we at first mined method association rules with reasonable constraints from the set of commit operations of a subject system.

Then, we inferred the patterns and *MMCG*s from the selected association rules. We describe our detection mechanism in the following paragraphs beginning with the formal definition of the association rule and related terminology.

A. Association Rule

An association rule [1] is an expression of the form X => Y where X is the antecedent and Y is the consequent. Each of X and Y is a set of one or more program entities. As we consider only methods in our experiment, the sets X and Y consist of methods only. The meaning of such a rule in our context is that if X gets changed in a particular commit operation, Y also has the tendency of getting changed in that commit operation.

We can determine the *confidence* or strength of a particular association rule by determining the *supports* of its constituent parts in the following way.

Support: Support is the probability of a method or a group of methods by which the method or method group appears (gets modified) in commit operations. The support of a method set X is calculated according to the following equation.

$$support(X) = \frac{C_X}{C} \quad (1)$$

Here C_X is the count of commits in which X appears and C is the total number of commit operations. Support of the expression X => Y is determined by *support (X Y)* in the following way.

$$support(X => Y) = support(XY) = \frac{C_{XY}}{C} \quad (2)$$

Here C_{XY} is the count of commits in which both X and Y appeared.

Confidence: Confidence is the conditional probability of an association rule X => Y. Confidence of this association rule determines the probability that Y will appear in a commit operation provided that X appears in that commit operation. We determine the confidence of X => Y in the following way.

$$confidence(X => Y) = \frac{support(X => Y)}{support(X)} = \frac{C_{XY}}{C_X} \quad (3)$$

In our experiment we consider only those rules where each of X and Y consist of a single method. Such a rule can be expressed as x => y where x and y are two different methods.

B. Constraints for Detecting MMCG

Suppose x, y, and z are three methods of a particular software system. Let us further assume that their co-change history obey the following two constraints.

(i) Each of the rules x => y and z => y has confidence 1.
(ii) Each of the rules y => x and y => z has a confidence of less than 1.

Looking at Fig. I we can see that the methods: $m1$, $m2$, and $m3$ co-changed following the above constraints. Here, the method $m2$ corresponds to y in the described constraints. These constraints have the following implications.

(1) According to the first constraint, whenever x modifies y gets a corresponding modification. Also, because of every change in z there is a corresponding change in y. However, the second constraint implies that there are some commits during the evolution where y received some changes but any one or both of x and z did not receive any change. Thus, the above two constraints guarantee that x, y and z have co-changed following the defined pattern. Here, y is the *MMCG*, and also y is likely to be logically coupled with both x and z.

(2) The constraints also confirm that the *MMCG*, y, will be changed in more commits compared to both x, and z during the evolution. Thus, these constraints have the potential to detect frequently modified *MMCG*s.

(3) As the constraints confirm that y will appear in more commits compared to both x and z, it is likely (not guaranteed) that y will have a longer lifetime (in term of how many commit operation it is remaining alive without getting deleted) compared to the others. Obviously, higher modification frequency cannot confirm higher longevity.

(4) As the *MMCG*s are likely to: (i) be coupled with multiple other methods, (ii) have higher modification frequency, and (iii) have higher longevity, *MMCG*s might be correlated with higher change proneness of a software system.

We have empirically evaluated each of these four implications in Section V (Experimental Results and Discussion).

C. MMCG Detection Steps

Step 1: In this step we identify the association rules of the form a => b from commit operations where a and b are two different methods and

1) a => b has confidence 1,
2) b => a has confidence less than 1, and
3) a and b got modified together in at least a predefined minimum number of commit operations C_{min}. If we denote the count of commits having both a and b by C_{ab}, we can specify this condition as, $C_{ab} \geq C_{min}$.

The third condition has been imposed because it might increase the possibility of the existence of an underlying relationship between a and b. It is difficult to choose a particular value for C_{min}. We detected *MMCG*s with four different values (1, 2, 3, 4) for C_{min} as described in Section V. For higher values of C_{min} we found no *MMCG*s for some subject systems (e.g. Each of Camellia, GreenShot, and MonoOSC). *Step-1* is performed with the help of Apriori algorithm[2]. Apriori is a frequent item-set mining algorithm which we have used to identify method pairs appearing in at least C_{min} commit operations.

Step 2: In this step we determine the *MMCG*s from the rules identified in *Step-1*. We identify those methods each of which appears as the consequent (the method at the right hand side of a rule selected in *Step-1*) of more than one association rules. Each of these methods is an *MMCG* according to our definition. We sort the *MMCG*s in non-increasing order of the count of their antecedents. For each *MMCG* we determine the set of its antecedents so that we can perform manual investigation on whether an *MMCG* is logically coupled with its antecedents. The details of our manual investigation are presented in Section V-A.

[2]Apriori algorithm. http://en.wikipedia.org/wiki/Apriori_algorithm

D. Related Terminology

For determining the confidences of association rules and supports of methods or method groups we need to determine the number of commit operations in which a particular method or a particular group of methods appeared. For this purpose we need to identify the method genealogies. Without method genealogies we cannot determine whether the same method or method group has appeared in multiple commit operations.

Method Genealogy: 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 a separate instance of this method. Method genealogy identifies all of these instances as belonging to the same method. Therefore, the total number of method genealogies extracted from the revisions of a system is equal to the number of methods created in different revisions of that system. A particular method genealogy corresponds to a particular method.

Methods Not Selected as *MMCG*s (MNSM): In our experiment we investigate only those methods that were modified during evolution. The methods that did not appear in the commit operations have not been analyzed, because these methods did not exhibit evolutionary coupling. After determining the method genealogies of a particular software system we identify those genealogies which have received some changes in their lifetime. By excluding the *MMCG*s from these genealogies we obtain the *MNSM*s.

IV. Experimental Setup and Steps

We implemented our proposed method using the Java programming language with MySQL as the backend database server. We applied our methodology to seven open source subject systems to determine the *MMCG*s and *MNSM*s. Table I lists the subject systems downloaded from an open source SVN repository[3]. The subject systems are diverse, varying in size, spanning six different application domains, and covering three different programming languages.

For a particular subject system we completed the following steps sequentially: **(1)** Preprocessing of source code, **(2)** Detection of methods along with file name, class name (for Java, and C#), package name (for Java), method name, signature, starting and ending line numbers from all the revisions of the subject system using Ctags[4] and storing the methods in the database, **(3)** Extraction of method genealogies following the methodology proposed by Lozano and Wermelinger [16], **(4)** Detection of clones using NiCad [9] and storing them in the database, **(5)** Locating the clones in the already detected methods using beginning and ending line information of clone fragments and methods, **(6)** Determining the changes between every two consecutive revisions using *diff* and locating these changes to the methods and clones, **(7)** Identification of *MMCG*s and *MNSM*s, and finally **(8)** Calculation of metrics. For the detailed description of the first six steps we refer the interested readers to our study [18]. We have described the identification process of *MMCG*s and *MNSM*s in Section III.

[3]Source Forge: http://sourceforge.net/

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

TABLE I
Subject Systems

	Systems	Domains	LOC	Revisions
Java	DNSJava	DNS Protocol	23,373	1635
	Carol	Game	25,092	1699
C	Ctags	Code Def. Generator	33,270	774
	Camellia	Multimedia	85,015	207
	QMail Admin	Mail Management	4,054	317
C#	GreenShot	Multimedia	37,628	999
	MonoOSC	Formats and Protocols	18,991	355

TABLE II
Method Counts

	Systems	MG	MGC	MMCG	MNSM	MMCG	MNSM
				C_{min} = 1		C_{min} = 2	
Java	DNSJava	3752	1244	493	751	128	1116
	Carol	5004	943	376	567	122	821
C	Ctags	865	362	91	271	14	348
	Camellia	307	206	83	123	45	161
	QMail Admin	120	59	40	19	22	37
C#	GreenShot	1378	393	102	291	15	378
	MonoOSC	604	164	60	104	9	155

MG = Method Genealogies
MGC = Method Genealogies that received some changes
MMCG = Methods appearing in Multiple Commit Groups
MNSM = Methods Not Selected as *MMCG*

We calculated the metrics *Life Span* (LS), and *Modification Occurrence Rate* (MOR) for measuring the longevity and change proneness of *MMCG*s and *MNSM*s. We calculated two other metrics: *Count of LOC Modified per Commit* (CLMC), and *Modification Count per Commit* (MCC) to quantify the change proneness of the source code of the entire software system in two different ways. We calculated these metrics to find a correlation between the proportion of *MMCG*s and the source code change proneness of the software systems. The details of these metrics are described in Section V.

V. Experimental Results and Discussion

We applied our methodology on each of the seven candidate systems and obtained *MMCG*s for four different values of C_{min} (mentioned in Section III-C): 1, 2, 3 and 4. However, the *MMCG*s obtained for higher values of C_{min} are obviously the subsets of the *MMCG*s obtained for lower values of C_{min}. We observe that for some subject systems (e.g. MonoOSC, Camellia, Ctags, Greenshot) the counts of *MMCG*s resulted for C_{min} = 3 or 4 are negligible compared to the counts of *MMCG*s obtained for C_{min} = 1 or 2. Our experimental result section consists of the results and statistics obtained for two C_{min} values: 1 and 2.

Table II provides some basic counts for each candidate system, specifically: the number of method genealogies (MG), the number of genealogies that received some changes (MGC), and the number of *MMCG*s and *MNSM*s for C_{min} = 1 and 2. The total number of *MMCG*s and *MNSM*s is equal to the number of method genealogies with changes, because we detected *MMCG*s considering those genealogies that received

some changes during the evolution. Inclusion of the methods that did not change during evolution is not necessary, because evolutionary coupling cannot be inferred from these methods.

In section III-B we mentioned four implications regarding the *MMCG*s detected using our proposed method co-change pattern. In the following four subsections we empirically evaluate these implications. We also investigate the effect of clones on method co-change.

A. Manual Investigation on the Detected MMCGs

In Section III-B we mentioned that the *MMCG*s detected according to our proposed pattern are likely to be logically coupled with multiple other methods. In this subsection we manually investigate whether the detected *MMCG*s are logically coupled with their respective antecedents. We have manually checked all the *MMCG* results obtained for two subject systems: Ctags and QMailAdmin. For the purpose of checking we devised our algorithm to obtain the following information for each *MMCG*.

(1) what are the antecedents of a particular *MMCG*

(2) in which commit operations the method pair consisting of an *MMCG* and an antecedent of it (as detected by our algorithm) have changed together. Obviously, according to our definition of the pattern, the *MMCG* has changed in all those commit operations where the antecedent has changed.

We analyzed the changes to the *MMCG* and the corresponding antecedents in each of those commit operations where they changed together. We wanted to determine whether the changes occurred in an *MMCG* and its antecedent are related. Our investigation regarding Ctags is given below.

(1) Investigation of Ctags: In case of C_{min} = 1, we obtained 91 *MMCG*s (Table II) among which 55 *MMCG*s (60% of the *MMCG*s) can be suggested to be logically coupled with some of their corresponding antecedents according to our manual investigation. The average number of logically coupled antecedents per *MMCG* (considering these 55 *MMCG*s) was five (highest = 8, lowest = 2). An example of a highly coupled *MMCG* is *processToken* (file: *c.c*) which is logically coupled with eight of its antecedents: $accessString$, $declToTagType$, $includeTag$, $qualifyCompoundTag$, $qualifyFunctionTag$, $tagLetter$, $tagName$, and $qualifyVariableTag$ in the same file. The *MMCG processToken* co-changed with each of these antecedents in different commit operations. In the commit on revision 242, these nine methods (*processToken* and its antecedents) co-changed. In this commit, a particular condition that checks for language CSharp was added in *processToken* and corresponding changes (e.g., addition of the same condition or corresponding statements) also took place to each of the eight antecedents. This can be easily verified by observing the changes in these methods comparing the instances of *c.c* file in revisions 242 and 243. We obtained five *MMCG*s in total each of which was found to be logically coupled with eight of their antecedents. However, our implementation sorts the *MMCG*s in non-increasing order of the count of their antecedents so that we can consider the first few highly coupled *MMCG*s for analyzing and possible restructuring to minimize their coupling with their antecedents.

The changes occurred between an *MMCG* and a corresponding antecedent can be divided into the following categories: (1) addition of string constants in an antecedent and addition of corresponding case statements in *MMCG*, (2) addition of corresponding case statements in both *MMCG* and its antecedent, (3) addition of string constants in an antecedent and addition of condition for that constant in *MMCG*, (4) addition of the same condition in both *MMCG* and its antecedent, (5) deletion of the same method calls from both *MMCG* and its antecedent, and (6) changes in the parameters of an antecedent and corresponding changes to *MMCG* because *MMCG* calls the antecedent. For C_{min} = 2, we obtained 14 *MMCG*s (Table II). We found 10 *MMCG*s (71.42% of *MMCG*s) that are logically coupled with some of their antecedents.

(2) Investigation of QMailAdmin: According to our investigation on QMailAdmin, the methods in this system are very highly coupled. For C_{min} = 1, we detected 40 *MMCG*s in total (Table II). Through manual analysis we selected 27 *MMCG*s (67.5% of the *MMCG*s) each of which is logically coupled with 11 antecedents on an average (highest = 15, lowest = 3). The changes between an *MMCG* and a corresponding antecedent can be divided into four categories: (1) renaming of the same variables in these methods, (2) parameter change to the same method calls in these methods, (3) deletion of the same method calls, and (4) replacement of the same method call by another method call in each of these two methods. We also calculated the percentage of *MMCG*s for C_{min} = 2. Among 22 *MMCG*s selected for C_{min} = 2, 17 (77.27% of the *MMCG*s) appeared to be logically coupled with multiple antecedents. Thus, we see that considering C_{min} = 1 we get a higher number of *MMCG*s that are logically coupled with multiple other methods (antecedents).

We have already mentioned that in case of C_{min} = 3 and 4, the number of detected *MMCG*s is negligible for some subject systems. We see that the percentage of true positives (*MMCG*s that are logically coupled with some of their antecedents) in case of C_{min} = 1 is smaller than the percentage of true positives obtained for C_{min} = 2. Intuitively, higher values of C_{min} increases the likelihood of an existing logical coupling between an *MMCG* and its corresponding antecedents. However, considering C_{min} = 1 our algorithm can retrieve a higher amount of *MMCG*s that are logically coupled with their antecedents. We observed that the number of *MMCG*s rapidly decreases with the increase of C_{min}.

A complete example of changes occurred to an *MMCG* and its antecedents: We present an example from QMailAdmin showing how an *MMCG* and two of its antecedents co-changed following our proposed pattern through out the evolution. For this example, *addusernow* is the *MMCG*, and *adduser* and *delusergo* are the antecedents. These methods belong to the same file *qmailadmin/user.c*.

The antecedent *adduser* was modified in three commits applied to the revisions 139, 143, and 148. The *MMCG addusernow* was also modified in these commits. In the first

commit two calls to the methods *count_users* and *load_limits* were deleted from *adduser*. The same change also happened to *addusernow* in the same commit. In the second commit, two calls to *exit(0)* in *adduser* were replaced by two return statements *return (142)* and *return (199)*. In the same commit, two corresponding calls in *addusernow* were also replaced by *return (142)* and *return (199)*. In the third commit, the parameter of the *get_html_text ()* was changed in the corresponding places of *adduser* and *addusernow*.

The antecedent *delusergo* changed in three commits applied to revisions 76, 143, and 148. The *MMCG addusernow* also had corresponding modifications in these commits. In the first commit the parameter of *call_hooks* method was changed in the corresponding places of *delusergo* and *addusernow*. In the second commit, a call to the *vclose* method was deleted from the corresponding locations of these methods. The third commit made a change similar to that described in the previous paragraph.

The ways the methods: *addusernow*, *delusergo*, and *adduser* got modified obviously indicate that *addusernow* is related to the functionalities associated with *delusergo* and *adduser*. Thus, *addusernow* is an *MMCG* that is logically coupled with two other methods (antecedents).

B. Investigation on the Longevity of MMCGs

In Section III-B we mentioned that the *MMCG*s detected using our proposed method co-change pattern are likely to live longer compared to the *MNSM*s. In this subsection we investigate the longevity of *MMCG*s and *MNSM*s.

Life Span (LS): Life span measures the longevity of a particular method in terms of the number of commits where it was alive. Each commit operation creates a new revision. Suppose a particular method m was created in revision R_c and disappeared (was deleted) after revision R_d. The life span of this method will be calculated using Eq. 4.

$$LS(m) = R_d - R_c + 1 \tag{4}$$

Here, $LS(m)$ is the *life span* of the method m.

As we extract the method genealogies examining all the revisions, we can easily determine the R_c and R_d of a particular method corresponding to a particular method genealogy. We have determined the average life spans for *MMCG*s and *MNSM*s according to the following equations.

$$ALS_{MMCG} = \frac{\sum_{m \epsilon MMCG} LS(m)}{|MMCG|} \tag{5}$$

$$ALS_{MNSM} = \frac{\sum_{m \epsilon MNSM} LS(m)}{|MNSM|} \tag{6}$$

Here, ALS_{MMCG} and ALS_{MNSM} are the average life spans of *MMCG*s and *MNSM*s respectively. $MMCG$ and $MNSM$ are respectively the sets of all *MMCG*s and *MNSM*s. We should mention that we are investigating only those methods that have received some changes in their life span. We separated these methods in two disjoint sets $MMCG$ and $MNSM$.

ALS_{MMCG} and ALS_{MNSM} for each of the subject systems are plotted in the bar graph of Fig. 2. We see that for each of the candidate systems the average life span of *MMCG*s is greater than that of *MNSM*s for both C_{min} = 1 and 2. Thus, *MMCG*s generally live longer than *MNSM*s. We wanted to determine whether the life spans of *MMCG*s are significantly longer than those of *MNSM*s. For this purpose we performed the following significance test.

Significance test regarding longevity: We performed the Mann-Whitney U-Test [17] on the observed life spans of *MMCG*s and *MNSM*s for each system. Test details are given in Table III. This table also contains the significance test results for another metric (MOR). We will describe this later.

Here, we should mention that for Mann-Whitney U-Tests the two sets of populations do not need to be normally distributed. A particular test gives us a U-value associated with a probability value (two tailed p-value). We have recorded only the p-values. A p-value smaller than 0.05 indicates a significant difference between the metric values of *MMCG*s and *MNSM*s. For each of the tests in Table III the sample sizes are the counts of *MMCG*s and *MNSM*s of a particular subject system.

According to the test regarding lifespan, *for most of the subject systems (all systems in case of C_{min} = 1, five systems out of seven in case of C_{min} = 2), life spans of* MMCG*s are significantly longer than life spans of MNSMs*. In other words, *MMCGs generally live significantly longer than MNSMs*.

C. Investigation on the Instability of MMCGs

By the term *instability* we mean the change proneness of methods (*MMCG*s and *MNSM*s). From our pattern definition and detection technique of *MMCG*s it is likely that the *MMCG*s will appear to receive more frequent changes compared to *MNSM*s. This is also mentioned in Section III-B. However, we wanted to investigate whether the instability of *MMCG*s is significantly higher than *MNSM*s. We investigate the following instability metric for this purpose.

Modification Occurrence Rate (MOR): Modification occurrence rate measures the number of modifications taking place to a method per commit operation. We consider modifications according to the definition given by Hotta et al. [14]. Just for clarification we can say that a single modification can affect several (one or more) consecutive lines. Suppose, 10 lines of a particular method has been modified in a particular commit operation. If these lines are consecutive (without any gap of unchanged lines among them), the count of modifications is one. Otherwise, the count of modifications is equal to the number of gaps among the modified lines plus one. For a particular method m we calculate its modification occurrence rate $MOR(m)$ according to the following equation.

$$MOR(m) = \frac{MC(m)}{LS(m)} \tag{7}$$

Here, $MC(m)$ is the count of modifications occurring in method m during its life span. We calculated the modification occurrence rates for *MMCG*s and *MNSM*s of the entire system according to the following equations.

$$MOR_{MMCG} = \frac{\sum_{m \epsilon MMCG} MC(m)}{\sum_{m \epsilon MMCG} LS(m)} \tag{8}$$

$$MOR_{MNSM} = \frac{\sum_{m \epsilon MNSM} MC(m)}{\sum_{m \epsilon MNSM} LS(m)} \tag{9}$$

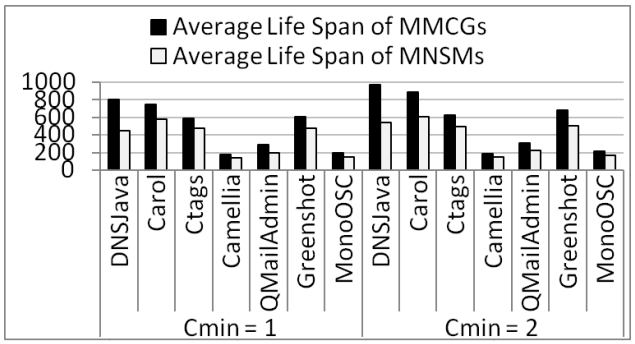

Fig. 2. Comparison regarding life span.

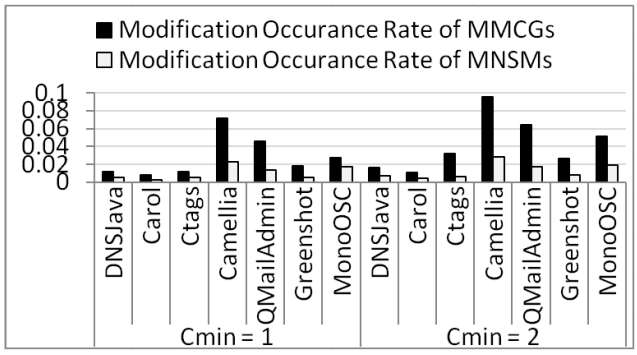

Fig. 3. Comparison regarding modification occurrence rates (MOR).

Here, MOR_{MMCG} and MOR_{MNSM} denote the modification occurrence rates of *MMCG*s and *MNSM*s respectively.

We compared the modification occurrence rates of *MMCG*s (MOR_{MMCG}) and *MNSM*s (MOR_{MNSM}) of each of the software systems by plotting the values of these two metrics in the graph of Fig. 3. We see that for each of the systems MOR_{MMCG} is much greater than MOR_{MNSM} (for each value of C_{min}). Thus, *the rate by which MMCGs receive modifications is always higher than the rate by which MNSMs receive modifications according to our subject systems.* This is expected considering our definition and detection technique of *MMCG*s. However, we wanted to determine whether *MMCG*s exhibit significantly higher instability compared to *MNSM*s.

Significance test regarding instability: For the purpose of the significance test we calculated the modification occurrence rate (MOR) of each of the *MMCG*s and *MNSM*s according to the equation Eq. 7. Then we performed the Mann-Whitney U-Tests on the MORs of *MMCG*s and *MNSM*s for each of the subject system. Based on the test results given in Table III we can say that *the modification occurrence rates of MMCGs is always significantly higher than those of MNSMs for the subject systems*.

D. Investigation on the Correlation of MMCGs with Change-proneness of Source Code

From Section V-A we have observed that, most of the *MMCG*s are logically coupled with multiple other methods. Also, we have seen that *MMCG*s live significantly longer and exhibit significantly higher instability compared to *MNSM*s. From this we suspected that presence of *MMCG*s can be an indication of higher change-proneness of source code. To

TABLE III
RESULTS OF MANN WHITNEY WILCOXON (MWW) TESTS

Systems	DnsJava	Carol	Ctags	Camellia	QMai Admin	Green Shot	Mono OSC
p-values regarding LS and MOR for C_{min} = 1							
p (LS)	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	<0.001	0.002
p (MOR)	< 0.001	< 0.001	< 0.001	< 0.001	0.007	< 0.001	0.035
p-values regarding LS and MOR for C_{min} = 2							
p (LS)	< 0.001	< 0.001	0.071	0.003	<0.001	0.033	0.209
p (MOR)	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	0.026

*Sample sizes are the counts of *MMCG*s and *MNSM*s for Cmin = 1 and 2
*Sample sizes are listed in Table II
p (LS) = Two tailed probability value for Life Span (LS)
p (MOR) = Two tailed probability value for MOR
*A probability value < 0.05 indicates a significant difference

TABLE IV
CORRELATION OF MMCG PROPORTION WITH SYSTEM INSTABILITY

Systems	DnsJava	Carol	Ctags	Camellia	QMai Admin	Green Shot	MonoOSC
C_{min} = 1, Correlation between PMMCG and CLMC							
PMMCG	39.63	39.87	25.13	40.29	67.79	25.95	36.58
CLMC	21.93	42.58	23.62	35.04	128.8	18.98	50.85
Pearson Correlation Co-efficient between PMMCG & CLMC = 0.9232							
C_{min} = 1, Correlation between PMMCG and MCC							
PMMCG	39.63	39.87	25.13	40.29	67.79	25.95	36.58
CMC	10.63	16.87	7.94	18.12	51.31	9.57	22.51
Pearson Correlation Co-efficient between PMMCG & CLMC = 0.9373							
C_{min} = 2, Correlation between PMMCG and CLMC							
PMMCG	10.29	12.94	3.87	21.84	37.29	3.82	5.49
CLMC	21.93	42.58	23.62	35.04	128.8	18.98	50.85
Pearson Correlation Co-efficient between PMMCG & CLMC = 0.8519							
C_{min} = 2, Correlation between PMMCG and MCC							
PMMCG	10.29	12.94	3.87	21.84	37.29	3.82	5.49
CMC	10.63	16.87	7.94	18.12	51.31	9.57	22.51
Pearson Correlation Co-efficient between PMMCG & CLMC = 0.8742							

PMMCG = Proportion of *MMCG*s
CLMC = Count of LOC Modified per Commit
CMC = Count of Modifications per Commit

quantify the change-proneness of source code we measured the following two metrics.

(i) Count of source code lines modified per commit operation (CLMC)
(ii) Count of modifications (as defined in Section V-C) occurred per commit operation (CMC)

We wanted to see the whether the proportion of *MMCG*s is correlated with source code change-proneness. Table IV shows the details of correlation. The strong correlation (indicated by the correlation co-efficients) of the proportion of *MMCG*s (*PMMCG*) with each of the two metrics (*CLMC*, and *CMC*) implies that *higher change proneness in a software system is an strong indicator of the existence of higher proportion of MMCGs in that system*. While calculating *CLMC* and *CMC* we considered only those commits where some source code was modified.

E. Investigation Regarding the Effect of Clones on Co-change

A common intuition regarding software clones is that clone fragments in the same clone class are likely to co-change,

because changes in one clone fragment might require corresponding changes to other clone fragments in the same clone class to maintain consistency among the fragments. We investigated the exact (Type 1) and near-miss clones (Type 2, and Type 3 clones) in our study. While Type 1 clones are the same code fragments disregarding the comments and indentations, Type 2 clones retain syntactic similarity. In general, Type 2 clones are created because of variable renaming and changing of data types. Type 3 clones are created because of addition, deletion, or modification of lines in Type 1 or Type 2 clones. We define the following two co-changes regarding clones for the purpose of describing our investigation.

Co-change of Clone fragments from the Same Class (*CCSC*): If two or more clone fragments belonging to the same clone class co-change (in a commit operation), then we term this co-change as a **Co-change of Clone fragments from the Same Class** (*CCSC*).

Consistency Ensuring Co-change to Clones (CECC): If two or more clone fragments from the same clone class co-change (in a commit operation) because, changes in one clone fragment require corresponding changes to the other clone fragment(s) to ensure consistency among the fragments, then we call this co-change a **Consistency Ensuring Co-change to Clones** (*CECC*). Intuitively, most of the *CCSC*s are *CECC*s.

Investigation Process: We detected three types of clones (Type 1, Type 2, and Type 3) from each of the revisions of a software system using the NiCad clone detector [9], because NiCad can detect each of these three clone types with high precision and recall [22]. NiCad provides clone results by separating them into different clone classes according to the similarity of the clone fragments. We reflected or located these clones in the already detected methods using the beginning and ending line numbers of the clone fragments and methods. From the previously stored information regarding the changes between two consecutive revisions, we can identify whether a particular change occurred inside a clone fragment of a method. Considering the rules obtained from *Step 1* of *MMCG Detection Steps* (Section III-C) we determined those rules each of which obeys the following constraints.

(1) The two methods in the rule contains clone fragments from the same clone class, and

(2) In a particular commit operation, these clone fragments (from the same class) of these two methods co-changed.

Such a rule is related to *CCSC*. According to the common intuition regarding clones, it is likely that a rule with these constraints will also be related to *CECC*. We calculated the following two proportions for each of the subject systems.

(1) Proportion of Rules Related to *CCSC* (PRRC): Suppose, the number of rules obtained from *Step 1* of *MMCG Detection Steps* (Section III-C) is R, and r of these R rules are related to *CCSC*. We determine, ***PRRC = (r × 100) / R***.

(2) Proportion of *MMCG*s Related to *CCSC* (PMRC): Suppose, the number of *MMCG*s detected from *Step 2* of *MMCG Detection Steps* (Section III-C) is N, and n of these *MMCG*s are also the consequents of some of the rules related to *CCSC*. So, each of these n *MMCG*s is also related to *CCSC*. In other words, each of these n *MMCG*s has clone fragment which has co-changed with another clone fragment contained in one (at least) of its antecedents and these two clone fragments (clone fragment in *MMCG* and clone fragment in its antecedent) belong to the same clone class. We determine the proportion of these n *MMCG*s related to *CCSC*. This proportion is, ***PMRC = (n × 100) / N***.

We show these proportions for C_{min} = 1 and 2 in Table V. According to this table, a considerable amount of rules (34.43 % for QMailAdmin, C_{min} = 2) can sometimes be related to *CCSC*. Moreover, a significant proportion of *MMCG*s (68.18 % for QMailAdmin, 63.28 % for DNSJava considering C_{min} = 2) can be related to *CCSC*.

Manual Examination: For QMailAdmin, we manually investigated: (1) how many of the rules related to *CCSC* are also related to *CECC* (Consistency ensuring co-change to clones), and (2) how many of the *MMCG*s related to *CCSC* are also related to *CECC* for C_{min} = 2. We found 126 rules and 15 *MMCG*s that are related to *CCSC*. According to our observation, each of these 126 rules and 15 *MMCG*s is also related with *CECC*. As an example we mention the *MMCG addusernow* and its antecedent *modusergo* (both in file *user.c*) that co-changed in the commit operation applied on revision 143. The cloned portions in these methods are Type 3 clones belonging to the same clone class. In the commit on revision 143, the same called methods *vclose()* and *exit(0)* were replaced by a single return statement *return (142)* in both clone fragments of these two methods. Thus, these two methods (*addusernow*, and *modusergo*) received consistency ensuring co-changes to their clone fragments.

From the above scenario we realize that a significant portion of the *MMCG*s can exhibit evolutionary coupling because of the consistency ensuring changes to the clones of the same clone class. Also, a considerable amount of the rules are sometimes created because of such consistency ensuring changes. Thus, clone refactoring can be helpful in minimizing co-changes as well as *MMCG*s. In other words, by refactoring clones we can minimize evolutionary coupling among methods. We also determined the counts of rules related to *CCSC*s for three types of clones to determine which type(s) of clones mostly affect co-changing of methods. These counts are shown in Table VI. According to this table, for most of the subject systems the rules are mostly related to Type 1 or Type 3 clones. Thus, in case of refactoring we should possibly focus on Type 1 and Type 3 clones.

F. Investigation Results and Discussion

We summarize our investigation results as follows.

(1) The *MMCG*s detected following the method co-change pattern are sometimes highly coupled (i.e. coupled with many other methods) according to our manual investigation. Thus, the investigated pattern as well as the algorithm for identifying this pattern have the capability of retrieving highly coupled methods from the method co-change history.

(2) The detected *MMCG*s live significantly longer and show significantly higher change proneness compared to the other

TABLE V
PROPORTIONS OF RULES AND MMCGS RELATED TO CCSC

Systems	DnsJava	Carol	Ctags	Camellia	QMai Admin	Green Shot	MonoOSC
Proportions considering C_{min} = 1							
PRRC	1.75	1.00	0.69	0.49	20.9	0.86	0.57
PMRC	32.86	20.47	7.69	18.07	55	7.84	6.66
Proportions considering C_{min} = 2							
PRRC	11.14	12.63	0	3.57	34.43	0	2.5
PMRC	63.28	32.79	0	6.67	68.18	0	11.11

PRRC = Proportion of rules related to *CCSC*
PMRC = Proportion of *MMCG*s related to *CCSC*

TABLE VI
COUNTS OF RULES RELATED TO THREE TYPES OF CLONES

Systems	DnsJava	Carol	Ctags	Camellia	QMai Admin	Green Shot	MonoOSC
Counts of rules considering C_{min} = 1							
Type 1	164	11	0	12	61	0	0
Type 2	0	0	2	6	53	7	0
Type 3	55	214	6	11	38	1	3
Counts of rules considering C_{min} = 2							
Type 1	78	6	0	7	45	0	0
Type 2	0	0	0	0	46	0	0
Type 3	24	120	0	0	35	0	2

methods not detected as *MMCG*s. The possible reason behind the higher change proneness of *MMCG*s is that most of them are highly coupled with other methods and higher coupling can cause increased modifications to coupled entities[1].

(3) Higher change proneness in the entire source code is a strong indicator of the presence of a higher proportion of *MMCG*s as well as higher coupling in the software system.

(4) A considerable amount of method co-changes occurs because of the consistency ensuring changes in the clone fragments belonging to the same clone class. Also, sometimes a significant portion of *MMCG*s exhibit evolutionary coupling because of such consistency ensuring changes.

As lower coupling is generally regarded as a desirable attribute (because higher coupling promotes changes), our primary goal of this research work was to automatically identify the highly coupled areas in a software system so that we can think of possible restructuring in those areas to minimize coupling. We believe that our proposed method co-change pattern and methodology for detecting this pattern as well as *MMCG*s are significant contributions towards achieving this goal. Also, as the detected *MMCG*s are the most unstable (change prone) as well as long-lived methods in a software system, detection and understanding of these methods along with their relationships is very important for a new maintenance programmer. The proportion of detected *MMCG*s can possibly be regarded as a quantifier of design-quality attribute of a software system. Higher proportion of *MMCG*s is possibly an indication of design deficiency in a software system. From the correlation Table IV we see that the subject system QMailAdmin appears to have the highest proportion of *MMCG*s for both C_{min} = 1 and 2. We also see that this system exhibits the highest change-proneness. From Table V we can see that the effect of cloning on the method co-changes in QMailAdmin is the highest. Thus, QMailAdmin can be regarded as an example of a badly designed system.

Finally, our clone analysis result suggests that refactoring of clones can help us in minimizing method co-changes (evolutionary coupling among methods). In the case of refactoring we can primarily target Type 1 and Type 3 clones, because the association rules are more related to these two types of clones compared to the Type 2 clones.

VI. THREATS TO VALIDITY

The study is based on the assumption that co-change correlates with logical coupling. This is well accepted in the community. However, there are systems for which the correlation is not that large. The method co-change pattern that we investigated excludes a portion of co-changes from consideration, because these co-changes do not follow the constraints that we defined for the pattern. However, we believe that the co-changed methods obtained following the constraints are more likely to be logically coupled compared to the other co-changed methods excluded by the constraints. Thus, our defined pattern is important for detecting highly coupled entities.

The sample size of our study might not be sufficient to draw a general conclusion on the characteristics of *MMCG*s and *MNSM*s. However, as our selected subject systems are of diverse size, application domains, and programming languages we believe that our observations regarding the *MMCG*s and *MNSM*s cannot be attributed to a chance. Thus, our study result is important from maintenance perspectives.

VII. RELATED WORK

Association rules introduced by Agarwal et al. [1] have been frequently used to find associated or co-changing program artifacts (also known as frequent itemsets) [6], [28].

Zimmermann et al. [28] used *association rules* to represent the co-change relationships among different program artifacts. They implemented a tool called *ROSE* for extracting association rules from software evolution history. The *ROSE* prototype could predict the files needing to be modified for a particular change request for 26% of the cases. Canfora et al. [6] proposed the use of the Granger causality test for determining whether a particular change occurring in a program artifact is consequentially related to some other changes occurring in other artifacts. Jafar et al. [15] performed a comprehensive study on macro co-changes considering file level granularity. They introduced two metrics *MCC* (macro co-changes), and *DMCC* (diphase macro co-changes) and using their proposed approach *Macocha* they detected how many files exhibit *MCC* and *DMCC*. Beyer [3] implemented a co-change visualization tool *CCVISU* for extracting the underlying clustering of artifacts in a software system analyzing the CVS log files. *CCVISU* can help us in understanding the relationships among software artifacts and providing helpful guidance of changes happening in the maintenance phase. Gall et al. [11] introduced an approach for discovering logical dependencies and change patterns among different program

modules by using the information in the release history of a system to minimize the structural problems in the system. D'Ambros et al. [10] implemented *evolution radar* to visualize module level and file level logical couplings.

In an empirical study Hassan and Holt [13] showed that historical co-change information can be used to help developers during the change propagation process. Zhou et al. [27] presented a Bayesian network based approach for predicting change coupling behaviour between source code artifacts.

We see that there are a great many studies on co-changeabilities of program artifacts. Our empirical study presented in this paper is different in the sense that we investigate a unique method co-change pattern that is able to detect highly coupled methods that are important from the perspective of program comprehension. We also investigated the effects of clones on method co-changes with an interesting outcome on how to minimize method co-changes.

VIII. Conclusion

In this paper we investigate a particular method co-change pattern which is capable of identifying particular methods (*MMCG*s) that are likely to be logically coupled with multiple other methods. We implement our technique for detecting the method co-change pattern as well as *MMCG*s. According to our empirical evaluation with rigorous manual investigation, the *MMCG*s detected according to our mentioned method co-change pattern are often logically coupled with multiple other methods, live significantly longer and exhibit significantly higher change-proneness compared to the other methods not detected as *MMCG*s. A possible cause behind this higher change proneness is that *MMCG*s are logically coupled with many other methods. Source code change-proneness of the entire software system is positively correlated with the proportion of *MMCG*s. Also, a considerable amount of method co-changes can occur because of the consistency ensuring changes in the clone fragments from the same clone class. A significant portion of *MMCG*s can exhibit evolutionary coupling because of such consistency ensuring changes.

We come to the conclusion that as lower coupling among program entities is always desirable[1], the highly coupled *MMCG*s detected by our technique are areas where it may be to restructure the system to minimize coupling as well as change-proneness of the system. Also, as the *MMCG*s exhibit significantly higher change-proneness and live significantly longer compared to other methods not selected as *MMCG*s, understanding of *MMCG*s along with their relationships should be given a high priority for a new maintenance programmer.

Observing our clone analysis results we believe that clone refactoring can help us in minimizing evolutionary coupling among methods. While refactoring we should primarily focus on Type 1, and Type 3 clones because these two types of clones are more related with method co-changes compared to Type 2 clones according to our study.

As future work we plan to develop a visualization tool that will help programmers in defining, detecting, and visualizing different method co-change patterns and will identify the highly coupled methods along with their relationships.

References

[1] R. Agrawal, T. Imieliski, A. Swami, "Mining association rules between sets of items in large databases", Proc. *ACM SIGMOD*, 1993, 22(2): 207 – 216.

[2] G. Antoniol, V. F. Rollo, and G. Venturi, "Detecting groups of co-changing files in cvs repositories", Proc. *IWPSE*. 2005, pp. 23 – 32.

[3] D. Beyer. "Co-change visualization", Proc. *ICSM*, 2005, pp. 89–92

[4] D. Beyer, A. E. Hassan, "Animated visualization of software history using evolution storyboards", Proc. *WCRE*, 2006, pp. 199–210.

[5] S. Bouktif, Y.-G. Gueheneuc, G. Antoniol, "Extracting changepatterns from cvs repositories", Proc. *WCRE*, 2006, pp. 221–230.

[6] G. Canfora, M. Ceccarelli, L. Cerulo, M. Di Penta, "Using multivariate time series and association rules to detect logical change coupling: An empirical study", Proc. *ICSM*, 2010, pp. 1–10.

[7] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories", Proc. *Metrics*, 2005, p. 29.

[8] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta, "An eclectic approach for change impact analysis", Proc. *ICSE*, 2010, pp. 163–166.

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

[10] M. D'Ambros, M. Lanza, M. Lungu, "Visualizing co-change information with the evolution radar", Proc. *TSE*, 2009, 35(5): 720 – 735.

[11] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history", Proc. *ICSM*, 1998, pp. 190–199.

[12] D. M. German, "An empirical study of fine-grained software modifications", Proc. *ESE*, 2006, 11(3): 369 - 393.

[13] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems", Proc. *ICSM*, 2004, pp. 284–293.

[14] 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.

[15] F. Jaafar, Y. Gueheneuc, S. Hamel, G. Antoniol, "An Exploratory Study of Macro Co-changes", Proc. *WCRE*, 2011, pp. 32–334.

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

[17] Mann-Whitney U-Test details: http://www.experiment-resources.com/mann-whitney-u-test.html

[18] M. Mondal, C. K. Roy, and K. A. Schneider, "Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems", Proc. *CASCON*, pp. 205 – 219, 2012.

[19] A. J. Offutt, M. J. Harrold, P. Kolte, "A software metric system for module coupling", *Journal of Systems and Software*, 1993, pp. 295–308

[20] M. Page-Jones, "The practical guide to structured systems design", YOURDON Press, New York, NY, 1980.

[21] Peter Weigerber, L. v. Klenze, M. Burch, and S. Diehl "Exploring evolutionary coupling in Eclipse", Proc. *OOPSLA workshop on Eclipse technology eXchange*, 2005, pp. 31 – 34.

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

[23] A. Vanya, R. Premraj, and H. v. Vliet, "Interactive Exploration of Co-evolving Software Entities", Proc. *CSMR*, 2010, pp. 260 – 263.

[24] A. Vanya, L. Hofland, S. Klusener, P. van de Laar, and H. van Vliet, "Assessing software archives with evolutionary clusters", Proc. *ICPC*, 2008, pp. 192 – 201.

[25] S. Wong, and Y. Cai "Generalizing evolutionary coupling with stochastic dependencies", Proc. *ASE*, 2011, pp. 293 – 302.

[26] A. T. T. Ying, G. C. Murphy, R. Ng, M. C. Chu-Carroll, "Predicting source code changes by mining change history", *TSE*, 2004, 30(9):574 – 586.

[27] Y. Zhou, M. Wursch, E. Giger, H. C. Gall, and J. Lu, "A bayesian network based approach for change coupling prediction", Proc. *WCRE*, 2008, pp. 27–36.

[28] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, "Mining version histories to guide software changes", Proc. *ICSE*, 2004, pp. 563–572.

[29] T. Zimmermann, S. Kim, E. James Whitehead Jr., A. Zeller, "Mining Version Archives for Co-changed Lines", Proc. *MSR*, 2006, pp. 72-75.