A Fine-Grained Analysis on the Evolutionary Coupling of Cloned Code

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

***Abstract*—Code clones are identical or similar code fragments in a code base. A group of code fragments that are similar to one another forms a clone class. Clone fragments from the same clone class often need to be changed together consistently and thus, they exhibit evolutionary coupling. Evolutionary coupling among clone fragments within a clone class has already been investigated and reported. However, a change to a clone fragment of a clone class may also trigger changes to non-cloned code as well as to clone fragments of other clone classes. Such coupling information is equally important for the proper management of clones during software maintenance. Unfortunately, there are no such studies reported in the literature. In this paper, we describe a large scale empirical study that we conduct to examine whether a clone fragment from a particular clone class exhibits evolutionary coupling with non-clone fragments and/or with clone fragments of other clone classes. Our experimental results on thousands of revisions of six diverse subject systems written in two programming languages indicate the presence of such couplings. We consider both exact and near-miss clones in our study. By analyzing the evolutionary couplings of a particular clone fragment from a particular clone class, we are able to predict its three types of co-change candidates with considerable accuracy in terms of precision and recall. These co-change candidates are: (1) non-clone fragments, (2) clone fragments from clone classes other than its own class, and (3) other clone fragments from its own clone class. Thus, we can improve existing clone tracking techniques so that they can also infer and suggest which non-clone fragments as well as which clone fragments from other clone classes might need to be co-changed correspondingly when modifying a clone fragment from a particular clone class.**

I. Introduction

If two or more code fragments in a code base are identical or similar to one another, we call them code clones [29]. A group (two or more) of identical or similar code fragments forms a clone class. Clones are mainly created because of frequent copy-paste activities performed by programmers during both software development and maintenance. Clones are of great importance during software maintenance. A great many studies have been conducted on the detection and analysis of code clones in software systems [2], [3], [11], [14]–[19], [23], [24], [33]. While a number of studies [11], [14], [16], [17] identify some positive impacts of code clones, there is strong empirical evidence [3], [18], [19], [23] of some negative impacts of clones on software evolution and maintenance. These negative impacts include late propagation [3], hidden bug propagation [3], unintentional inconsistencies [3], and high change-proneness [23]. Focusing on these negative impacts researchers emphasize that it is important for efficient management of code clones to support clone detection, refactoring, and tracking [7], [22], [30].

A number of techniques and tools [28], [31] for detecting clones already exist. Clone refactoring refers to the task of merging several clone fragments into a single one [22]. However, refactoring of all clone fragments in a software system is impractical [15]. There can be situations where clone refactoring is impossible, however, the clone fragments need to be updated consistently [6], [15]. Thus, clone tracking is important. Clone tracking helps us to update the clone fragments in a clone class consistently to avoid unintentional inconsistencies and late propagation [7]. Our investigation in this research focuses on clone tracking.

Usually clone tracking means remembering the clone fragments of a particular clone class so that while changing a particular clone fragment from this class in the future, we can look at the other clone fragments in this class and can decide whether these other clone fragments also need to be changed together (i.e., co-changed) to ensure consistency. A number of clone tracking techniques and tools exist [7], [8], [12], [20], [34]. A recent study focused on predicting and ranking co-change candidates that are the other clone fragments from the same clone class when modifying a particular clone fragment from that clone class [25].

The basic idea or assumption behind each of the existing studies and tools regarding clone tracking is that a change in a particular clone fragment can affect other clone fragments in the same clone class. The research ignores the fact that a change in a clone fragment can also affect non-clone fragments as well as clone fragments from other clone classes. In other words, while changing a particular clone fragment from a particular clone class, we might also need to co-change some non-clone fragments, as well as some clone fragments from the other clone classes correspondingly. In this case, an efficient clone tracking system should not only focus on tracking (i.e., remembering) the other clone fragments from the same class but also focus on tracking probable co-change candidates that are: (1) non-clone fragments, as well as (2) clone fragments from other clone classes.

In this research, we investigate whether a clone fragment from a particular clone class exhibits evolutionary coupling with non-clone fragments as well as with clone fragments from other clone classes. In the presence of such couplings, it is possible to predict which non-clone fragments, and also which clone fragments from other clone classes might need to be co-changed correspondingly while changing a particular clone fragment from a particular clone class. To the best of our knowledge, our study is the first to investigate these couplings. Here, we should note that by the term 'corresponding changes' we mean 'related changes'. If two code fragments changed together by receiving related changes, we say that they co-changed correspondingly. We define and explain corresponding changes in Section II.

TABLE I. RESEARCH QUESTIONS

SL	Research Question
1	Do clone fragments have co-change tendencies with non-clone fragments? If so, are the changes corresponding?
2	Do clone fragments from different clone classes have co-change tendencies? If so, are the changes corresponding?
3	Can evolutionary coupling help us predict: (i) which non-cloned fragments, and (ii) which clone fragments from other clone classes might need to be co-changed while changing a particular clone fragment from a particular clone class?

Through our investigation on thousands of revisions of six diverse subject systems covering two programming languages we answered three important research questions listed in Table I. According to our experimental results and analysis considering both exact and near-miss clones:

> *A clone fragment can exhibit evolutionary coupling not only with the other clone fragments in its own class but also with non-clone fragments as well as with clone fragments from other clone classes.*

According to our manual analysis, a change occurring to a clone fragment in a particular clone class might require related changes to non-clone fragments as well as to clone fragments from other clone classes.

> *By analyzing the evolutionary couplings of a particular clone fragment from a particular class, we can predict: (1) its non-clone co-change candidates with overall recall and precision of 27.07% and 16.5%, (2) its co-change candidates that are clone fragments from the other clone classes with overall recall and precision of 30.74% and 20.67%, (3) its co-change candidates that are the other clone fragments from the same clone class with overall recall and precision of 80.46% and 74.24%, and finally (4) all three co-change candidates with overall recall and precision of 33.73% and 22%.*

Existing clone tracking techniques only focus on tracking, predicting, and ranking of clone fragments from the same clone class. However, our experimental results indicate that for proper management of code clones, clone tracking techniques should also be able to infer and track change couplings (i.e., evolutionary couplings) between clone and non-clone fragments and also, between clone fragments from different clone classes so that when a programmer changes a particular clone fragment from a particular clone class, a clone tracker can help them by suggesting: (1) possible non-clone fragments, as well as (2) possible clone fragments from other clone classes that might also need to be co-changed (i.e., changed together) correspondingly.

A recent study conducted by Kagdi et al. [13] shows that consideration of finer granularity (such as method) in detecting co-change candidates using evolutionary coupling results in very low precision and recall. Kagdi et al. [13] achieved at best 9% precision and 28% recall. In a previous study Zimmermann et al. [37] achieved 26% precision and 15% recall. However, detecting evolutionary coupling considering finer granularity is important, because it can help programmers pin point co-change candidates while making changes to a software system [26], [27]. In our study we investigate evolutionary coupling considering code fragment granularity (clone and non-clone fragments) which is even finer than method granularity. However, our main focus is on detecting co-change candidates for clones. Considering these issues and the previously reported precision and recall values we believe that our precision and recall results in detecting co-change candidates for clones are interesting and significant.

The rest of the paper is organized as follows. Section II describes the terminology, Section III elaborates on the methodology, Section IV answers the research questions on the basis of the experimental results, Section V mentions possible threats to validity, Section VI discusses the related work, and Section VII concludes the paper mentioning future work.

II. TERMINOLOGY

Types of clones. We conduct our experiment considering exact (Type 1) and near-miss clones (Type 2 and Type 3 clones). As is defined in the literature [29], if two or more clone fragments in a particular clone class are exactly the same disregarding comments and indentations, these clone fragments are called exact clones of one another (i.e., Type 1 clones). Type 2 clones are syntactically similar code fragments. In general, Type 2 clones are created from Type 1 clones because of renaming variables or changing data types. Type 3 clones are mainly created because of additions, deletions, or modifications of lines in Type 1 or Type 2 clones.

Corresponding changes. Let us assume that two code fragments have co-changed (changed together) in a particular commit operation. If the changes are related such that changes in one code fragment required changes to the other fragment to ensure consistency between them, then we say that the changes to these two code fragments are corresponding changes. Here, a code fragment can be a clone fragment from a particular clone class or a non-clone fragment (defined in Section III-B). When two code fragments co-change correspondingly, they receive related changes such as addition or deletion of the same statements in both code fragments. In Sections IV-A and IV-B, we provide examples of corresponding changes that we found by manually investigating the evolution history of our subject systems.

Evolutionary coupling. During the evolution of a software system if two or more program entities (such as files, classes, methods) appear to change together (i.e., co-change) frequently (i.e., in many commits) then we say that these entities exhibit evolutionary coupling. It is likely that these entities are related and a future change to any one of these entities will accompany corresponding changes to the other entities. Evolutionary coupling is also known as change coupling or logical coupling [9]. By mining and analyzing evolutionary coupling we can discover the underlying relationships among program entities in a software system [37]. Evolutionary coupling helps us predict co-change candidates (i.e., other entities that might also need to be changed together) while changing a particular entity [37]. In our research, we apply the concept of evolutionary coupling to discover the underlying relationships : (1) among clone fragments in the same clone class, (2) among clone fragments from different clone classes, and (3) among clone and non-clone fragments to predict all possible co-change candidates while changing a particular clone fragment from a particular clone class.

Association Rule. Evolutionary coupling has been identified using association rules [1]. An association rule [1] is an expression of the form $X => Y$ where X is the antecedent and

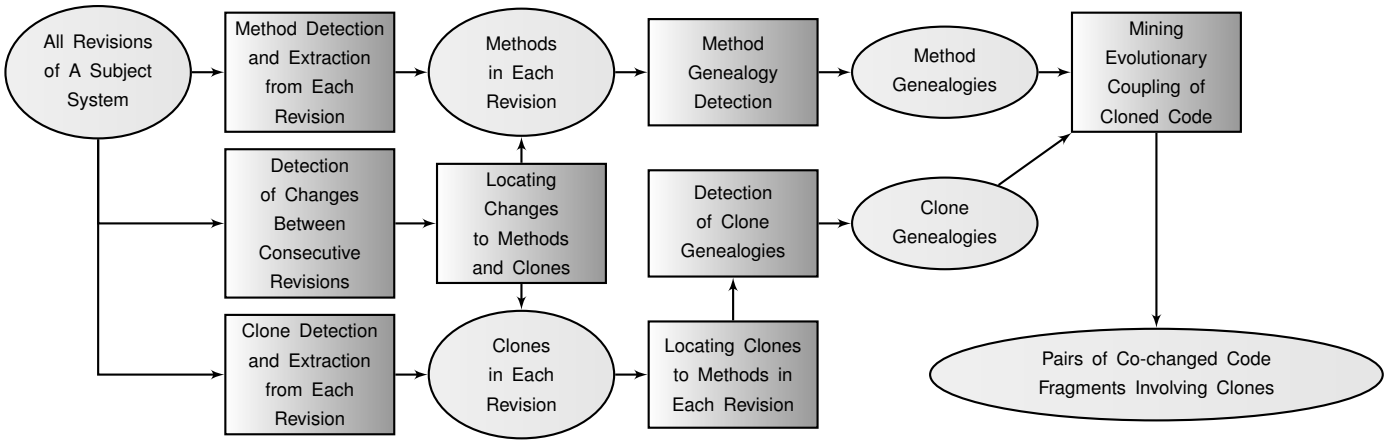

Fig. 1. The experimental steps in detecting evolutionary coupling (i.e., logical coupling or change coupling) of cloned code of a subject system

TABLE II. SUBJECT SYSTEMS

System	Language	Domain	LOC	Revisions
Ctags	C	Code Def. Generator	33,270	774
QMailAdmin	C	Mail Management	4,054	317
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Manager	45,515	1545

Revisions = Number of revisions investigated

Y is the consequent. Each of X and Y is a set of one or more program entities. The meaning of such a rule in our context is that if X gets changed in a particular commit, Y also has the tendency being changed in that commit. We determine the *support* and *confidence* of a rule in the following way.

Support and Confidence. As defined by Zimmermann et al. [37], *support* is the number of commits in which an entity or a group of entities changed together. Consider an example of two entities *E1* and *E2*. If *E1* and *E2* have ever changed together, we can assume two association rules, $E1$ => $E2$ and $E2$ => $E1$, from them. Suppose, *E1* was changed in four commits: 2, 5, 6, and 10 and *E2* was changed in six commits: 4, 6, 7, 8, 10, and 13. So, *support(E1) = 4* and *support(E2) = 6*. However, *support(E1, E2) = 2*, because *E1* and *E2* co-changed in two commits: 6 and 10. Support of a rule is determined as follows.

$$support(X \Rightarrow Y) = support(X, Y) \quad (1)$$

Where (X, Y) is the union of X and Y. Thus, *support(X => Y) = support(Y => X)*. From the above example, *support(E1 => E2) = support(E2 => E1) = support(E1, E2) = 2*.

Confidence of an association rule, $X \Rightarrow Y$, determines the probability that Y will change in a commit operation provided that X changed in that commit operation. We determine the confidence of $X \Rightarrow Y$ in the following way.

$$confidence(X \Rightarrow Y) = support(X, Y)/support(X) \quad (2)$$

From the above example of two entities, *confidence (E1 => E2) = support(E1, E2) / support(E1) = 2 / 4 = 0.5* and *confidence(E2 => E1) = 2 / 6 = 0.33*.

Higher values of support and confidence indicate stronger change coupling (i.e., evolutionary coupling) among the entities in an association rule. A detailed description of how we mine evolutionary coupling will be presented in Section III-B.

III. METHODOLOGY

Table II lists the six open source subject systems that we investigate in our study. We consider all the revisions (as noted in Table II) beginning with the first one for each of the systems.

A. Experimental Steps

We first download all the revisions as noted in Table II for all the subject systems from their open-source SVN repository[1]. Then, for each system we perform eight experimental steps as shown by the rectangles in Fig. 1 for mining evolutionary coupling of cloned code. These steps are: (1) Method detection and extraction from each of the revisions using CTAGS [2], (2) Detection and extraction of code clones from each revision using the NiCad clone detector [4], (3) Detection of changes between every two consecutive revisions using *diff*, (4) Locating these changes to the already detected methods as well as clones of the corresponding revisions, (5) Locating the code clones detected from each revision to the methods of that revision, (6) Detection of method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [19], (7) Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy, and (8) Mining the evolutionary coupling of cloned code. For the details of the first seven steps we refer the interested readers to our earlier work [21]. We will describe the eighth step in Section III-B.

Detecting the method-genealogy for a particular method involves identifying each instance of that method in each of the revisions where the method was alive. By detecting the genealogy of a method, we can determine how it changed during evolution. We detect clone genealogies by locating the clones detected from each revision to the already detected methods of that revision. The genealogy of a particular clone fragment also helps us determine how it evolved through the commits. We assign unique IDs to the method genealogies and clone genealogies to recognize them across revisions. Here, we should also note that we use NiCad [4] for detecting clones because it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [31], [32]. Using NiCad we detect block clones including both exact (Type 1) and near-miss (Type 2, Type 3) clones of a minimum size of 5 LOC with 20% dissimilarity threshold and blind renaming of identifiers. These settings (explained in detail in our earlier work [21]) are considered standard [30]. In the following subsection we describe how we mine evolutionary couplings for cloned code.

[1]Open source SVN repository. http://sourceforge.net/

[2]CTAGS: http://ctags.sourceforge.net/

B. Mining Evolutionary Coupling

By analyzing the evolution history of a subject system, we determine pairs of co-changed code fragments that can be categorized into the following categories.

- **Same-Class-Pair:** *Each pair in this category consists of two clone fragments, CF1 and CF2, that belong to the same clone class and co-changed (i.e., changed together) in one or more commit operations. If two or more clone fragments from a particular clone class ever co-changed in a particular commit, we determine all possible pairs from these co-changed clone fragments.*
- **Different-Class-Pair:** *Each pair in this category consists of two clone fragments, C1F1 and C2F1, that belong to two different clone classes and co-changed in one or more commit operations.*
- **Clone-Non-clone-Pair:** *A pair in this category consists of two code fragments, CF and NF, such that CF is a clone fragment from a particular clone class and NF is a non-clone fragment and these two code fragments co-changed in at least one commit operation.*

We provide a very simple example explaining how we form the pairs of co-changed code fragments. Let us assume that six code fragments changed together in a particular commit operation. These are: C1F1, C1F2, C1F3, C2F1, NF1, and NF2. The first three fragments are clone fragments belonging to the clone class C1. The clone fragment C2F1 belongs to the clone class C2. The remaining code fragments NF1, and NF2 are non-clone fragments. From these code fragments we form 14 pairs in total. Three pairs: (C1F1, C1F2) (C1F1, C1F3), and (C1F2, C1F3) are *Same-Class-Pairs*. Each of the three pairs: (C1F1, C2F1), (C1F2, C2F1), and (C1F3, C2F1) is a *Different-Class-Pair*. Eight pairs are *Clone-Non-clone-Pairs*. These are: (C1F1, NF1), (C1F2, NF1), (C1F3, NF1), (C2F1, NF1), (C1F1, NF2), (C1F2, NF2), (C1F3, NF2), and (C2F1, NF2). We examine each of the commit operations and determine the pairs of co-changed code fragments. A particular pair may appear more than once. We count the number of commits a pair appears.

In this experiment, we consider clone and non-clone fragments that reside in methods. The code fragments in a particular pair may reside: (1) in the same method, or (2) in two different methods. We consider both cases in our study. We determine the percentages of both *Same-Method-Pairs* (i.e., the pairs each consisting of code fragments from the same method) and *Different-Method-Pairs* in each of the above three categories and show the percentages for each subject system in Table IV. From this table we can see that most of the pairs in the two categories: *Different-Class-Pair*, and *Clone-Non-clone-Pair* consist of code fragments from different methods. We also see that *Same-Method-Pairs* mainly exist in the first category *Same-Class-Pair*. Moreover, for three subject systems: jEdit, Carol, and Jabref the percentage of *Same-Method-Pairs* in this category is very low. Although the proportion of *Same-Method-Pairs* is generally very low, they are important. From our manual investigation on our subject systems, we see that some methods are very long containing even more than 100 lines of code. Several clone fragments can be scattered in such a long method. Remembering as well as consistent updating of all the clone fragments in such a method might often be difficult without automatic support. So providing automated support for tracking couplings between code fragment pairs even within the same method is useful. However, not all methods are very long, so tracking information might be just overhead. One might even argue that although the couplings inferred from the *Same-Method-Pairs* are important for long methods, such couplings could possibly affect the overall statistics regarding evolutionary coupling in our research. However, from Table IV we again see that such couplings are of very low proportion in the two categories *Different-Class-Pair*, and *Clone-Non-clone-Pair*. As our main focus in this research is on detecting and analyzing evolutionary coupling between clone fragments from different clone classes (i.e., *Different-Class-Pairs*), and between clone and non-clone fragments (i.e., *Clone-Non-clone-Pairs*), we believe that our reported results are not affected by the couplings inferred from the *Same-Method-Pairs*.

TABLE III. NUMBER OF THE PAIRS OF CO-CHANGED CODE FRAGMENTS IN DIFFERENT CATEGORIES

	Carol	Jabref	Freecol	jEdit	Ctags	QMail.
Same-Class-Pairs	1081	367	626	101	77	53
Different-Class-Pairs	5201	4365	3041	6273	798	712
Clone-Non-clone-Pairs	10248	17963	8074	17626	1141	1562

TABLE IV. PERCENTAGE OF THE PAIRS CONSISTING OF CODE FRAGMENTS FROM THE SAME METHOD OR DIFFERENT METHODS

	Same-Class-Pairs		Different-Class-Pairs		Clone-Non-clone-Pairs	
	SM	DM	SM	DM	SM	DM
Ctags	51.95	48.05	10.15	89.85	3.68	96.32
QMailAdmin	54.72	45.28	11.24	88.76	2.24	97.76
Freecol	62.30	37.70	15.23	84.77	1.99	98.01
jEdit	24.75	75.25	1.05	98.95	0.37	99.63
Carol	4.63	95.37	5.06	94.94	1.01	98.99
Jabref	38.42	61.58	8.43	91.57	1.12	98.88

SM = % of pairs each consisting of code fragments from the same method
DM = % of pairs each consisting of code fragments from different methods

Mondal et al. [25] conducted an empirical study where they investigated evolutionary coupling among clone fragments from the same clone class only and thus, they only mined pairs of code fragments belonging to the first category (*Same-Class-Pairs*) defined above. Thus, we believe that our study provides further insight into the evolutionary coupling of cloned code.

We already mentioned that after detecting clones we locate them in the methods. A clone fragment is recognized by its starting and ending line numbers. However, non-clone fragments require explanation. We define a non-clone fragment in the following way.

A non-clone fragment. We consider a method m. If m contains one or more clone fragments, then the remaining code in this method (i.e., excluding the clone fragments) is considered as a non-clone fragment. Moreover, if m contains no clone fragments, then the whole method is considered a non-clone fragment. A fully cloned method does not contain a non-clone fragment.

For each pair of code fragments obtained in each of the above three categories, we determine how many times (i.e., in how many commit operations) the constituent code fragments co-changed. The XML files containing the pairs of code fragments (i.e., of three categories) that we detect from our subject systems are available on-line[3]. In such a file, we show the followings for each pair of code fragments: (1) The methods containing these code fragments, (2) The number of times the code fragments co-changed, (3) The commits where

[3]The XML Files: http://goo.gl/YkAY8V

Fig. 2. Comparison regarding the percentage of association rules

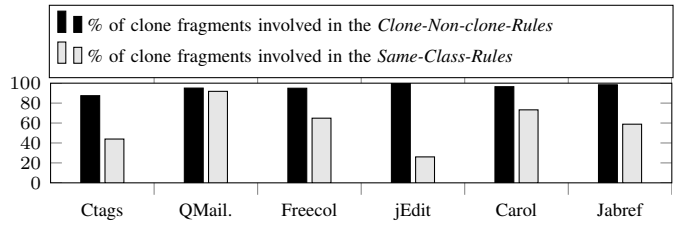

Fig. 3. Comparison regarding the percentage of clone fragments

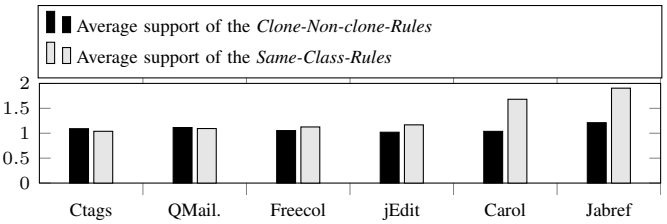

Fig. 4. Comparison regarding average support of the association rules

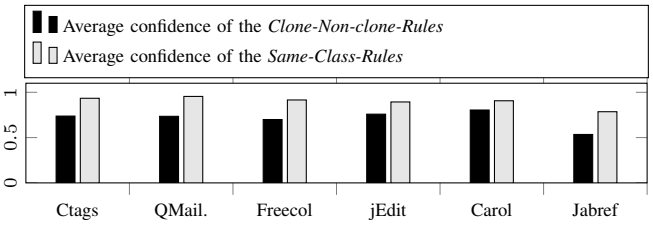

Fig. 5. Comparison regarding average confidence of the association rules.

they co-changed, and (4) The types of the code fragments indicating whether they are clone or non-clone fragments.

IV. Experimental Results and Discussion

We apply our implemented system on each of the six subject systems listed in Table II and determine the pairs of co-changed code fragments of each of the three categories mentioned in Section III-B. The number of pairs in each category is reported in Table III. In the following subsections we answer our research questions by analyzing these pairs.

A. *RQ 1: Do clone fragments have co-change tendencies with non-clone fragments? If so, are the changes corresponding?*

Rationale. Answering this question is important from the perspective of software maintenance. A recent study [25] shows that clone fragments from the same clone class have tendencies of co-changing correspondingly. The existing clone tracking techniques [7], [8], [12], [20], [34] help us determine co-change candidates from the same clone class. However, there is no study on whether a clone fragment has tendencies of co-changing correspondingly (Section II) with non-clone fragments as well as with clone fragments from other clone classes rather than its own clone class. If such tendencies are present, then a clone tracker should also be able to infer and store these so that while changing a particular clone fragment from a particular clone class, it can not only suggest possible co-change candidates (clones) from the same clone class but also can suggest: (1) possible non-clone fragments, as well as (2) possible clone fragments from other clone classes that might need to be co-changed correspondingly with the particular clone fragment. In this research question, we investigate whether clone fragments have co-change tendencies with non-clone fragments, and if so, whether the changes are corresponding changes. Co-change tendencies between clone fragments from different clone classes have been investigated in the second research question (*RQ 2*).

Methodology. For answering this research question, we determine association rules from the pairs of co-changed code fragments that we mine by analyzing the evolution history of a software system. Here we should note that association rules have been used to represent co-change tendencies between program artifacts [22], [37]. From a particular pair of code fragments (*CF1*, *CF2*) we can determine two association rules, *CF1* => *CF2* and *CF2* => *CF1*. These two rules have the same support value. However, their confidence values can be different. We determine association rules with support and confidence values from all the pairs of co-changed code fragments of a subject system. From these rules, we determine two sets of rules as defined below.

- ***Clone-Non-clone-Rules:*** *The Set of Rules each consisting of a Clone fragment and a Non-clone fragment.*
- ***Same-Class-Rules:*** *The Set of Rules each consisting of Clone fragments from the Same clone class.*

Mondal et al. [22] report that clone fragments within a clone class have co-change tendencies. We wanted to see whether co-change tendencies between clone and non-clone fragments are comparable to the co-change tendencies between clone fragments from the same clone class. We performed the following four comparisons on the sets *Clone-Non-clone-Rules* and *Same-Class-Rules*.

- Comparison between the percentage of rules in the two sets with respect to all rules in the system.
- Comparison between the percentage of clone fragments involved in the rules in the two sets with respect to all the clone fragments involved in all the rules in the system.
- Comparison between the average supports of the rules in the two sets.
- Comparison between the average confidences of the rules in the two sets.

These four comparisons for each of the subject systems are presented in Figures 2, 3, 4, and 5 respectively. Fig. 2 and 3 indicate that the percentage of rules as well as the percentage of clone fragments involved in the rules in the first set (*Clone-Non-clone-Rules*) are generally higher compared to the other set *Same-Class-Rules*. From Fig. 4 and 5 we see that for most of the subject systems the average support and confidence of the *Clone-Non-clone-Rules* are smaller than the corresponding values of the *Same-Class-Rules*. However, the average supports as well as the average confidences from the two sets are comparable to each other for each of the systems.

From these comparisons we can state that *a clone fragment from a particular clone class not only can co-change with other clones in the same class but also can co-change with non-clone fragments. Moreover, the percentage of clone fragments having co-change tendency with non-clone fragments can be*

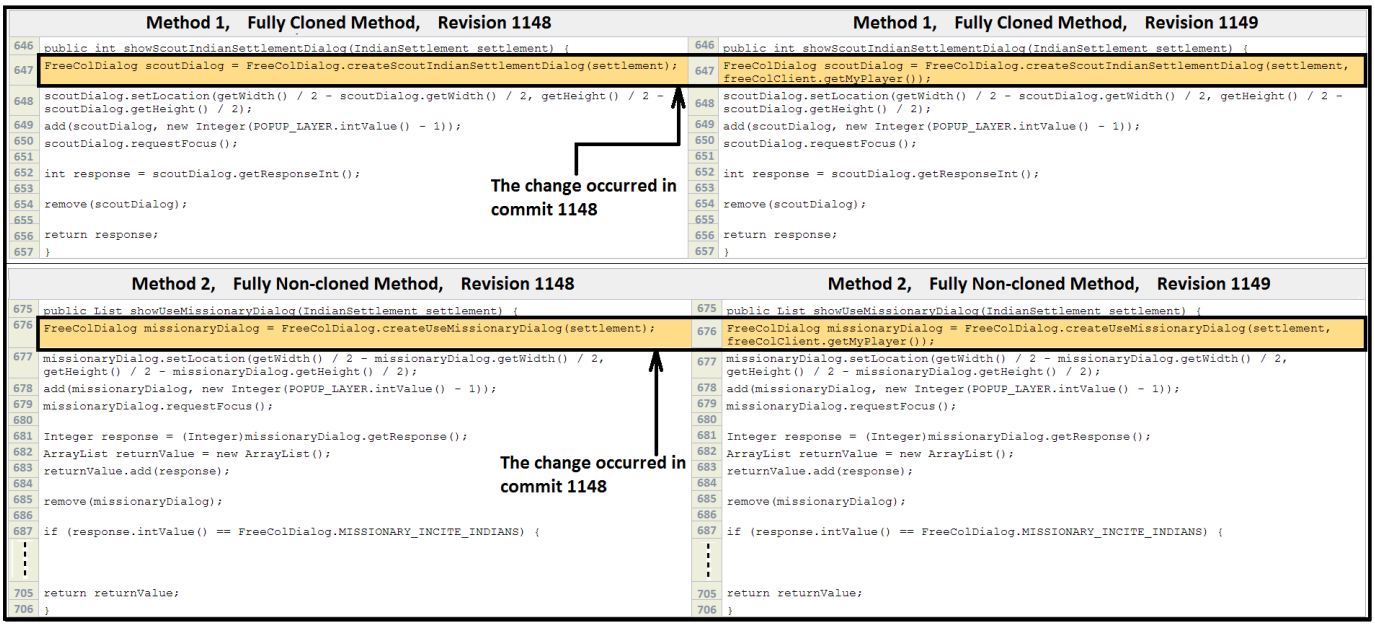

Fig. 6. Example of a corresponding co-change of a fully cloned method and a fully non-cloned method in the commit operation applied on revision 1148 of our candidate system Freecol.

higher compared to the percentage of clone fragments that co-change from the same clone class. Thus, we can say that the co-change tendencies between clone and non-clone fragments are comparable to the co-change tendencies between clone fragments from the same clone class.

As clone fragments exhibit co-change tendencies with non-clone fragments, we were interested to investigate whether they co-change correspondingly or not. We perform our investigation in the following way.

For each of the subject systems we select the *Clone-Non-clone-Rules* and sort them in descending order of the support values of the rules, because higher support value indicates a higher possibility that the changes to the code fragments were related. We manually examined the co-change history of the constituent code fragments of each of the top 20 rules from each of the subject systems. We determine whether the code fragments co-changed by receiving related changes or not (i.e., whether they co-changed correspondingly or not). According to our manual analysis on 120 rules from all candidate systems, *in the case of 85.83% rules (103 rules), the constituent code fragments (a clone fragment and a non-clone fragment) co-changed correspondingly.*

We also manually investigate 20 rules with the lowest support (support = 1) from each of the subject systems. If a rule has the lowest support, then it means that the constituent code fragments in the rule co-changed only once during the whole period of evolution. We manually investigated the lowest support rules, because we wanted to see whether the constituent code fragments even in these rules co-changed correspondingly. Promisingly, in case of around 51.25% of these lowest support rules the two constituent code fragments (i.e., a clone and a non-clone fragment) co-changed correspondingly. Thus, whether a clone fragment and a non-clone fragment co-change frequently or not, their changes can be corresponding. So, co-change detection (whether frequent or infrequent) and tracking of clone and non-clone fragments should be given a great importance not only for improving the clone tracking techniques but also for better analysis of change impacts [13].

As an example of a corresponding co-change, we mention two methods: Method 1 (*showScoutIndianSettlementDialog*, start line = 646, end line = 657) and Method 2 (*showUseMissionaryDialog*, start line = 675, end line = 706) from the same file *src/net/sf/freecol/client/gui/Canvas.java* of revision 1148 of our subject system Freecol. While Method 2 is a fully non-cloned method, Method 1 is a fully cloned method. Method 1 has three other clones in the same clone class (as of Method 1) in revision 1148. These clones also remain in the same file.

Method 1 and Method 2 co-changed in three commits applied on revisions: 1148, 1216, and 1341. In each of these commits these methods received related changes. Fig. 6 presents the changes occurred to these methods in the commit on revision 1148. We see that the second line of each method was changed in the same way. An extra parameter *freeColClient.getMyPlayer()* was added beside the previous parameter *settlement*. The other three clone fragments residing in the same clone class as of Method 1 did not change in this commit.

During our manual investigation, we also identify the major change types that a clone fragment and a non-clone fragment receive while co-changing correspondingly. The dominating changes occurred to them include: **(1)** addition of the same parameter to the similar called methods (Fig. 6 shows an example), **(2)** addition, deletion, or changing of parameters to the same called methods, **(3)** addition or deletion of the same condition, **(4)** changing of the same condition in the same way, **(5)** changing variable names in similar ways, **(6)** removal and/or addition of the same method calls, and **(7)** addition of named constants.

> ***Answer to RQ 1:*** *According to our analysis and discussion we come to the conclusion that a clone fragment from a particular clone class not only can co-change with the other clone fragments in the same class, but also can co-change correspondingly (i.e., co-change by receiving related changes) with non-clone fragments.*

The existing clone tracking techniques [7], [8], [12], [20], [34] only focus on remembering clone fragments from the

Fig. 7. Comparison regarding the percentage of association rules.

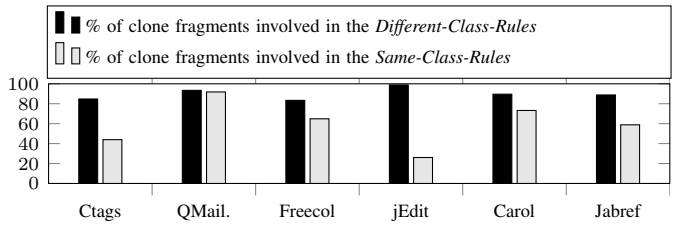

Fig. 8. Comparison regarding the percentage of clone fragments.

same clone class. However, our findings from this research question imply that clone tracking techniques should also be able to infer and remember co-change tendencies between clone and non-clone fragments so that while changing a particular clone fragment from a particular clone class, a programmer can also get automatic suggestions about which non-clone fragments might need to be co-changed correspondingly with that particular clone fragment.

B. ***RQ 2:*** *Do clone fragments from different clone classes have co-change tendencies? If so, are the changes corresponding?*

In this research question we investigate whether clone fragments from different clone classes have co-change tendencies, and if so, whether they co-change correspondingly. We perform our investigation in a similar way as described in *RQ 1*. We at first determine the following two sets of association rules from all the rules in a subject system.

- ***Different-Class-Rules:*** *The Set of Rules each consisting of Clone fragments from Different clone classes.*
- ***Same-Class-Rules:*** *The Set of Rules each consisting of Clone fragments from the Same clone class.*

We perform four comparisons between these two sets as was done in *RQ 1*. The comparison graphs regarding: the percentage of association rules, the percentage of clone fragments involved in the rules, and the average support and confidence of the rules are presented in Figures 7, 8, 9 and 10. Fig. 7 and 8 imply that the percentage of rules as well as the percentage of clone fragments involved in the rules in the first set *Different-Class-Rules* are generally higher compared to the corresponding percentages of the other set (*Same-Class-Rules*). From Fig. 9 and 10 we see that for most of the subject systems, the average support and confidence of the *Different-Class-Rules* are smaller than the corresponding values of the *Same-Class-Rules*. However, the average supports as well as the average confidences regarding the two sets are most of the cases very near to each other and thus, they are comparable.

From the above comparisons we can state that *clone fragments from different clone classes can have co-change tendencies and these tendencies are comparable to the co-change tendencies between clone fragments from the same clone class. Moreover, the percentage of clone fragments showing co-change tendencies from different clone classes can even be higher compared to the the percentage of clone fragments each having co-change tendencies with the other clone fragments in its own class.*

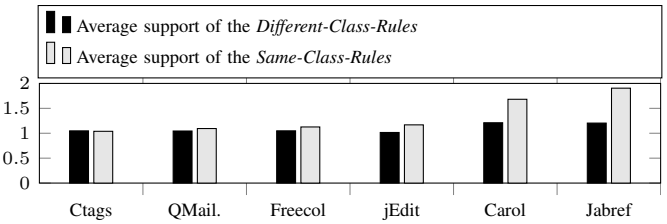

Fig. 9. Comparison regarding average support of the association rules.

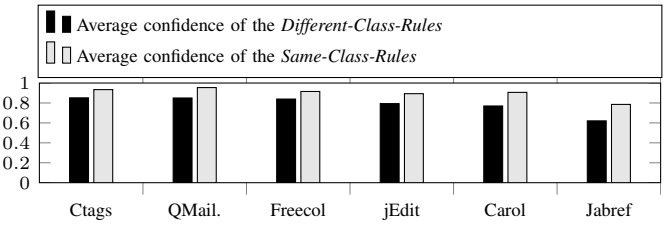

Fig. 10. Comparison regarding average confidence of the association rules.

As clone fragments from different clone classes have co-change tendencies, we were interested to investigate whether they co-change correspondingly. We perform manual investigation in a similar way as described in *RQ 1*. For each of the subject systems, we select all the rules between clone fragments from different clone classes, sort the rules in decreasing order of their support values, and then manually investigate the top 20 rules to determine whether the constituent clone fragments in each of the rules co-changed correspondingly. According to our analysis, *in the case of around 71.43% of the rules, the two constituent clone fragments from two different clone classes co-changed correspondingly*. We also investigate 20 rules with the lowest support value from each of the subject systems as we did in *RQ 1* and observe that in the case of around 45.94% of these rules the changes to the clone fragments were corresponding. Thus, whether clone fragments from different clone classes co-change frequently or not, they can receive corresponding changes. Therefore, detection as well as tracking of clone fragments that co-change from different clone classes (whether having strong co-change tendencies or not) are important for better clone management.

As an example of a corresponding co-change, we mention two clone fragments from two clone classes from our subject system Carol. These clone fragments belong to two different source code files: *File 1*[4] and *File 2*[5]. They co-changed in four commit operations applied on revisions: 36, 51, 54, and 58. At revision 36, the start and end lines of these clone fragments are (1) 101 (start line), 108 (end line) in *File 1* and (2) 138, 154 in *File 2* respectively. The changes occurred to the clone fragments in each of the four commits are related according to our manual investigation. For example, in the commit on revision 36, each of these fragments dealt with throwing and catching exceptions. In the commit on revision 51, similar *if-statement*s were added to these two clone fragments.

During manual investigation, we also examined which types of changes mainly occurred to the two clone fragments (from two different classes) when they co-changed correspond-

[4]File 1: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java
[5]File 2: carol/src/org/objectweb/carol/rmi/multi/MultiPRODelegate.java

ingly. We found all those changes that we have reported while answering *RQ 1*. The additional change types that frequently occurred during this investigation include: **(1)** changing the names of the same called methods. **(2)** addition or deletion of the same try/catch blocks, **(3)** addition or deletion of the same statements, and **(4)** addition or deletion of similar if-blocks.

> ***Answer to RQ 2.*** *From our analysis and discussion we can come to the conclusion that clone fragments from different clone classes can have tendencies of co-changing correspondingly.*

From our investigation we believe that a clone tracker should not only focus on tracking and predicting co-change candidates from the same clone class but also should focus on inferring co-change tendencies between clone fragments from different clone classes so that while changing a particular clone fragment from a particular clone class it can also suggest possible co-change candidates that are the clone fragments from other clone classes rather than its own clone class.

C. ***RQ 3:*** *Can evolutionary coupling help us predict: (i) which non-cloned fragments, and (ii) which clone fragments from other clone classes might need to be co-changed while changing a clone fragment from a particular clone class?*

Rationale. Answering this research question is the primary goal of our research work. From our answers to the previous two research questions we understand that a clone fragment from a particular clone class can correspondingly co-change with non-clone fragments as well as with clone fragments from the other clone classes. In this research question we investigate whether we can infer these evolutionary couplings to predict which non-clone fragments as well as which clone fragments from the other clone classes might need to be co-changed while changing a particular clone fragment from a particular clone class. However, we also investigate the predictability of co-change candidates from the same clone class as was studied by Mondal et al. [25]. Finally, we show a comparative scenario about predicting all three types of co-change candidates for a particular clone fragment.

Here, we should note that Mondal et al. [25] investigated evolutionary coupling of clone fragments within a clone class in order to predict co-change candidates from the same clone class only. Thus, we believe that our study provides further insight into the evolutionary coupling of cloned code.

For predicting co-change candidates we use a technique which is a variant of *n-fold cross validation technique*[6]. In this technique, the commits are examined from the very beginning one. Prediction of co-change candidates in a particular commit c is done by analyzing the evolution history in the past commits (i.e., the commits from 1 to $c-1$). Zimmermann et al. [37] used such a technique and it complies with the underlying philosophy of evolutionary coupling. We describe the technique in the following way.

Prediction Methodology. We consider a particular commit c where one or more clone fragments were changed. Some non-clone fragments might also be changed in this commit. We consider a particular clone fragment *CF* that changed in this commit c. As we automatically examine this commit, we determine which other code fragments co-changed with *CF* in this commit. However, our goal is to determine how accurately we can predict these other code fragments that co-changed with *CF* in commit c by analyzing the evolutionary coupling exhibited by *CF* during the past commits 1 to c-1. These other code fragments that actually co-changed with *CF* can be of three types. These are: (1) non-clone fragments, (2) clone fragments from the other clone classes rather than the clone class of *CF*, and (3) some other clone fragments from the same clone class as of *CF*.

From the past commits 1 to c-1 we determine all possible pairs of co-changed code fragments as is described in Section III-B. From these pairs we select those pairs each of which contains *CF*. From these selected pairs we determine the other code fragments beside *CF*. These other code fragments are the *suggested co-change candidates* for *CF* in commit c, because each of these other code fragments co-changed with *CF* at least once (i.e., exhibited evolutionary coupling with *CF*) during the past commits 1 to c-1. *Suggested co-change candidates* can also be of three types as we mentioned in the previous paragraph. If we see that a *suggested co-change candidate* actually co-changed with *CF* in commit c, then we can say that our prediction system has correctly predicted this candidate to be a co-change candidate for *CF* in the commit operation c.

We determine which of the *suggested co-change candidates* actually co-changed with *CF* in commit c. These are the candidates (i.e., code fragments) that are correctly predicted by our prediction system to be the co-change candidates for *CF*. We call these candidates the *correctly predicted co-change candidates* for *CF* in commit c. *Correctly predicted co-change candidates* can also be of three types as we mentioned in the previous two paragraphs. We can easily understand that the higher the number of *correctly predicted co-change candidates*, the higher is the accuracy of our prediction system. We determine the prediction accuracy of our implemented system in terms of precision and recall calculated using the following equations.

$$Precision = |CPCCS| * 100 \ / \ |SCCS| \quad (3)$$

$$Recall = |CPCCS| * 100 \ / \ |ACCS| \quad (4)$$

Here, *CPCCS*, and *SCCS* are respectively the sets of *correctly predicted co-change candidates*, and *suggested co-change candidates* for *CF* in commit c. *ACCS* is the set of candidates (i.e., code fragments) that actually co-changed with *CF* in the commit operation c.

Considering each of the clone fragments (*CF*) changed in each of the commit operations of a particular subject system, we determine the overall precision and recall in predicting each of the three types of co-change candidates separately.

TABLE V. PRECISION AND RECALL (IN PERCENTAGE) IN PREDICTING DIFFERENT CO-CHANGE CANDIDATES FOR CLONE FRAGMENTS

	Carol	Jabref	Freecol	jEdit	Ctags	QMailAdmin	Overall
Prediction of non-clone co-change candidates							
Recall	15.36	35.47	20.09	9.48	46.43	49.31	27.07
Precision	9.59	22.27	20.83	4.055	45.81	62.59	16.5
Prediction of co-change candidates that are clone fragments from different classes							
Recall	44.84	32.71	17.65	8.13	38.97	18.88	30.74
Precision	41.18	20.73	20.46	3.17	15.80	25.70	20.67
Prediction of co-change candidates that are clone fragments from the same clone class							
Recall	88.32	77.03	53.59	64.15	40	55.56	80.46
Precision	81.93	67.27	58.99	45.95	33.33	83.33	74.24
Prediction of all co-change candidates							
Recall	46.96	37.48	23.13	9.96	43.32	42.57	33.73
Precision	38.97	24.18	23.74	4.15	25.89	52.75	22

[6]http://en.wikipedia.org/wiki/Cross-validation_(statistics)

We also determine the overall combined precision and recall in predicting all three types of co-change candidates. These precision and recall results are reported in Table V.

From Table V we see that for most of the subject systems except Ctags, the precision and recall in predicting co-change candidates that are clones from the same clone class are the highest ones. However, the precision and recall in predicting non-cloned co-change candidates as well as in predicting co-change candidates that are clone fragments from other clone classes are also considerable for most of the systems except jEdit. Finally, looking at the overall (considering all subject systems) precision and recall results in the last column of Table V we realize that we can predict each of the three types of co-change candidates for clones with considerable accuracy.

> ***Answer to RQ 3.*** *From our analysis and discussion we can state that by analyzing evolutionary coupling of cloned code we can predict: (i) which non-clone fragments, and (ii) which clone fragments from other clone classes might need to be co-changed while changing a particular clone fragment from a particular class with considerable accuracy in terms of precision and recall.*

Our investigation and analysis in this research question indicates that the existing clone tracking techniques should not only focus on tracking (or remembering) and predicting co-change candidates from the same clone class, but also should be able to infer and store evolutionary coupling between clone and non-clone fragments as well as between clone fragments from different clone classes so that while making changes to a particular clone fragment from a particular clone class, a programmer can get automatic suggestions about its all three types co-change candidates (non-clone fragments, clone fragments from other clone classes, and other clone fragments from its own clone class). Our implemented prediction system can help the existing clone trackers to predict these all three types of co-change candidates with considerable accuracy in terms of precision and recall.

V. Threats to the Validity

We used the NiCad clone detector [4] for detecting clones. For different settings of NiCad, the statistics that we present in this paper might be different. Wang et al. [35] defined this problem as the *confounding configuration choice problem* and conducted an empirical study to ameliorate the effects of the problem. However, the settings that we have used for NiCad are considered standard [30] and with these settings NiCad can detect clones with high precision and recall [31], [32].

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the prediction of co-change candidates for clones. However, our candidate systems were diverse variety in terms of application domains, sizes and revisions. Thus, we believe that our findings are important and can help us to better manage code clones in particular can help build a better clone tracking system.

VI. Related Work

A great many studies have already been conducted on the detection, evolution [2], [15], [33], impact analysis [3], [11], [14], [16]–[19], [23], [24], and maintenance [8], [12], [20], [22], [34] of code clones. Although there are some positive impacts [11], [14], [16], [17] of cloning on both software development and maintenance, a number of studies [3], [18], [19], [23] have shown empirical evidence of some strong negative impacts of clones on software evolution. Focusing on the negative impacts of clones, software researchers have emphasized on properly maintaining code clones through clone refactoring [22] or tracking [7], [8], [12], [20], [34]. As clone refactoring is not always possible [15], tracking of clones becomes very important for better software maintenance. As our research work is focused on clone tracking, we discuss the existing clone tracking techniques and studies.

The most related study on clone tracking was conducted by Duala-Ekoko and Robillard [8]. They introduced the concept of *clone region descriptor*. On the basis of this concept they proposed a technique for tracking clones in evolving software. They implemented a tool called 'CloneTracker' [7] as an Eclipse plug-in for tracking clones. The tool provides support for two tasks: (1) change notifications, and (2) simultaneous editing of clones. After modifying a particular clone fragment *CF* tracked by CloneTracker, the programmer is notified about the other clone fragments in the same clone class that contains *CF*, because these other clone fragments from the same class might need to be co-changed correspondingly with *CF*. However, CloneTracker cannot suggest which non-cloned fragments as well as which clone fragments from different clone classes might also need to be co-changed correspondingly with *CF*. Our research focuses on automatically identifying these two categories of co-change candidates (i.e., non-clone fragments, and clone fragments from other clone classes) which are ignored by existing studies, techniques, and tools.

Jablonski and Hou [12] developed a tool called *CReN* to track copy-paste code clones and support consistent renaming of identifiers. Miller and Myer [20] proposed a technique for simultaneous editing in multiple clone fragments in the same clone class to minimize the task of repetitive editing. They implemented their technique in a text editor called *LAPIS*. There is also another clone tracking tool called *Codelink* developed by Toomin et al. [34]. However, none of these existing clone trackers is capable of suggesting possible non-clone co-change candidates as well as possible co-change candidates that are clone fragments from the other clone classes while changing a particular clone fragment from a particular class.

In a recent study Mondal et al. [25] conducted an empirical study on predicting and ranking of co-change candidates that are the other clone fragments from the same class while changing a particular clone fragment from that class. However, this study neither focused on predicting non-clone co-change candidates nor on predicting co-change candidates that are clone fragments from the other clone classes when changing a clone fragment from a particular class. Our study is different in the sense that we analyze the evolutionary coupling of cloned code to predict these two categories of co-change candidates (non-clone candidates, and clone fragments from other clone classes) that are ignored by the existing studies.

From our discussion above we believe that our study presented in this paper is important and unique. Our experimental results indicate that a clone fragment from a particular clone class not only can exhibit evolutionary coupling with the other clone fragments from the same clone class, but also can exhibit evolutionary coupling with non-clone fragments as well as with clone fragments from the other clone classes. Our implemented system can automatically infer these evolutionary couplings by analyzing clone evolution history. We believe that our findings have the potential to assist in the better management of code clones by strengthening existing clone trackers.

VII. Conclusion

In this research we primarily focus on investigating whether a clone fragment from a particular clone class exhibits evolutionary coupling with non-clone fragments as well as with clone fragments from the other clone classes. The existing studies and clone tracking techniques have ignored these couplings and only focused on evolutionary coupling between clone fragments within a clone class. Our experimental results on thousands of revisions of six diverse subject systems covering two different programming languages and considering both exact and near-miss clones indicate the presence of these couplings. We implement a prediction system that can infer the evolutionary couplings of the clone fragments in a software system and use these couplings to predict their future co-change candidates. According to our experimental results we can state that while changing a particular clone fragment from a particular clone class, we can predict its all three types of co-change candidates (non-clone fragments, clone fragments from other clone classes rather than its own class, and other clone fragments from its own class) with considerable accuracy in terms of precision and recall.

Considering the recent studies on evolutionary coupling, our reported precision and recall results (focused on predicting co-change candidates for clones only) are promising and are of significant importance. According to our experimental results and analysis, we can improve existing clone trackers so that they can also suggest possible non-clone co-change candidates as well as possible co-change candidates that are clone fragments from other clone classes when making changes to a clone fragment of a particular clone class. Thus, our study has the potential to assist in software maintenance.

As future work we plan to incorporate our clone co-change prediction technique as a plug-in for the Eclipse IDE to provide developers with continuous suggestions about co-change candidates during both software development and maintenance.

References

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

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

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

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

[5] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code", *IEEE Trans. Software Engineering*, 2002, 28(7):654 – 670.

[6] Cross Cutting Concerns: http://en.wikipedia.org/wiki/Cross-cutting_concern.

[7] E. Duala-Ekoko, and M. P. Robillard, "CloneTracker: Tool Support for Code Clone Management", Proc. *ICSE*, 2008, pp. 843 – 846.

[8] E. Duala-Ekoko, and M. P. Robillard, "Tracking Code Clones in Evolving Software", Proc. *ICSE*, 2007, pp. 158 - 167.

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

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

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

[12] P. Jablonski, and D. Hou, "CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE", Proc. *Eclipse Technology Exchange at OOPSLA*, 2007.

[13] H. Kagdi, M. Gethers, D. Poshyvanyk, M. L. Collard,"Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code", Proc. *WCRE*, 2010, pp. 119 – 128.

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

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

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

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

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

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

[20] R. C. Miller, and B. A.Myers. "Interactive simultaneous editing of multiple text regions.", Proc. *USENIX 2001 Annual Technical Conference*, 2001, pp. 161 – 174.

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

[22] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic Ranking of Clones for Refactoring through Mining Association Rules", Proc. *CSMR-WCRE*, 2014, pp. 114 - 123.

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

[24] M. Mondal, C. K. Roy, and K. A. Schneider, "An Empirical Study on Clone Stability", *ACM SIGAPP Applied Computing Review*, 2012, 12(3): 20 – 36.

[25] M. Mondal, C. K. Roy, and K. A. Schneider, "Prediction and Ranking of Co-change Candidates for Clones", Proc. *MSR*, 2014, pp. 32 – 41.

[26] M. Mondal, C. K. Roy, and K. A. Schneider, "Insight into a method co-change pattern to identify highly coupled methods: An empirical study", Proc. *ICPC*, 2013, pp. 103 – 112.

[27] M. Mondal, C. K. Roy, and K. A. Schneider, "Improving the detection accuracy of evolutionary coupling by measuring change correspondence", Proc. *CSMR-WCRE*, 2014, pp. 358 – 362.

[28] D. Rattan, R. Bhatia, M. Singh, "Software Clone Detection: A Systematic Review", *Information and Software Technology*, 2013, 55(7): 1165 – 1199.

[29] C. K. Roy, "Detection and analysis of near-miss software clones", Proc. *ICSM*, 2009, pp. 447 – 450.

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

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

[32] C. K. Roy, J. R. Cordy, "A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools", Proc. *Mutation*, 2009, pp. 157 – 166.

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

[34] M. Toomim, A. Begel, and S. L. Graham. "Managing duplicated code with linked editing", Proc. *IEEE Symposium on Visual Languages and Human Centric Computing*, 2004, pp. 173 – 180.

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

[36] M. F. Zibran, and C. K. Roy, "Conflict-aware Optimal Scheduling of Code Clone Refactoring", *IET Software*, 2013, 7(3): 167 – 186.

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