Automatic Identification of Important Clones for Refactoring and Tracking

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

***Abstract*—Code cloning is a controversial software engineering practice due to contradictory claims regarding its impacts on software evolution and maintenance. While a number of studies identify some positive aspects of code clones, there is strong empirical evidence of some negative impacts of clones too. Focusing on the issues related to clones researchers suggest to manage code clones through detection, refactoring, and tracking. However, all clones in a software system are not suitable for refactoring or tracking. Thus, it is important to identify which clones we should consider for refactoring and which clones should be considered for tracking. In this research work we apply the concept of evolutionary coupling to identify clones that are important for refactoring or tracking. By mining software evolution history, we determine and analyze constrained association rules of clone fragments that evolved following a particular change pattern called *Similarity Preserving Change Pattern* and are important from the perspective of refactoring and tracking. According to our investigation with rigorous manual analysis on thousands of revisions of six diverse subject systems covering two programming languages, overall 13.20% of all clones in a software system are important candidates for refactoring, and overall 10.27% of all clones are important candidates for tracking. Our implemented system can automatically identify these important candidates and thus, can help us in better maintenance of code clones in terms of refactoring and tracking.**

I. Introduction

If two or more code fragments in a code-base are identical or similar to one another, we call them code clones [28]. Two or more identical or similar code fragments form a clone class. Clones are mainly created because of frequent copy-paste activities performed by programmers during both software development and maintenance [31].

Clones are of great importance from the perspective of software maintenance and evolution [31]. A great many studies have been conducted on the detection and analysis of code clones in software systems [2], [5], [10], [16], [17], [19], [20], [23]. While a number of studies [10], [17], [18] identify some positive aspects of code clones, there is strong empirical evidence [5], [13], [19], [20], [23] of some negative impacts of clones on software evolution and maintenance. These negative impacts include late propagation [5], hidden bug propagation [5], and unintentional inconsistencies [5], [13]. Moreover, cloned code has higher change-proneness than non-cloned code [23]. Focusing on the issues related to clones researchers emphasize that it is important for efficient management of code clones to support clone detection, refactoring, and tracking [24], [29].

A number of techniques and tools [27], [29] for detecting clones already exist. Clone refactoring refers to the task of merging several clone fragments into a single one [24], [37]. However, refactoring of all clone fragments in a software system is impractical [16]. There can be situations where clone refactoring is impossible, however the clone fragments need to be updated consistently. Clone tracking (remembering all clone fragments in a particular clone class) is the most appropriate management technique for such clone fragments. Moreover, some clone fragments evolve independently, and therefore, we do not need to consider them for refactoring or tracking. Focusing on these issues we believe that for proper management of code clones we should emphasize on the following two important matters: **(1)** Identification of clones that are important for refactoring, and **(2)** Identification of clones that are important for tracking.

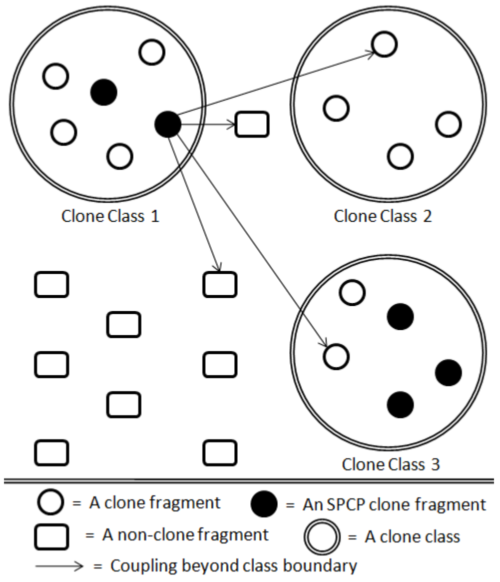


Fig. 1. Example of an SPCP clone fragment having relationships beyond its class boundary

In our previous study [24] we defined a particular clone change pattern called *Similarity Preserving Change Pattern* (SPCP) and proposed a mechanism for detecting all those clones in a software system that evolved following this pattern. We call these clones the *SPCP clones*. The clone fragments that do not follow this pattern either evolve independently or rarely change during evolution. Thus, these non-SPCP clone fragments cannot be important candidates for management. We proposed that the SPCP clone fragments might be important refactoring candidates.

However, an SPCP clone fragment from a particular clone class might have change couplings (i.e., evolutionary coupling relationships) with non-clone fragments and also, with clone fragments from other clone classes rather than its own class as shown in Fig 1. The figure shows that an SPCP clone fragment in *Clone Class 1* has couplings with four code fragments beyond its class boundary. Two code fragments are clone fragments from two other clone classes *Clone Class 2*, and *Clone Class 3*. The remaining two code fragments are non-clone fragments. Removal of this SPCP clone fragment might have ripple change effects[1] on these four code fragments remaining outside of *Clone Class 1* and also, might negatively affect their future evolution. Thus, according to our consideration, such an SPCP clone fragment having relationships across its class boundary should not be considered for removal through refactoring. Rather, these should be important candidates for tracking, because they evolve by maintaining consistency not only with other SPCP clones in their own clone classes but also with non-clone fragments as well as with clone fragments from other clone classes beyond their class boundaries. We should track these clone fragments along with their cross-boundary relationships so that we can update them (i.e., the SPCP clone fragment as well as the code fragments beyond its class boundary) consistently in future. However, if we want to refactor such an SPCP clone fragment, then we must analyze its relationships beyond its class boundary so that removal of it does not leave the related code fragments (beyond class boundary) in an inconsistent state and does not negatively affect the future evolution of these related code fragments.

Thus, it is important to automatically identify SPCP clone fragments that have couplings beyond their class boundaries so that we can discard these from consideration while taking refactoring decision and consider them as important candidates for tracking and future change prediction. Focusing on this important issue, in this paper we automatically extract and analyze the evolutionary coupling (i.e., change coupling) of SPCP clones and identify those SPCP clones each of which has change coupling with non-clone fragments and/or with clone fragments from other clone classes rather than its own clone class. Finally, we separate the whole set of SPCP clones into two disjoint subsets: **(1)** one contains the cross-boundary SPCP clones (i.e., the SPCP clones having cross boundary relationships), and **(2)** the other contains the non-cross-boundary SPCP clones. We perform an in-depth empirical study on both cross-boundary and non-cross-boundary SPCP clones.

According to our investigation considering both exact (Type 1) and near-miss (Type 2, Type 3) clones in thousands of revisions of six diverse subject systems covering two programming languages (Java, and C) we answer four research questions listed in Table I and come to the following decisions.

(1) The cross-boundary SPCP clones should be considered as the most important candidates for tracking. Removal of these clone fragments without taking proper care of their cross-boundary relationships might negatively affect the future evolution of the related code fragments. We should track cross-boundary SPCP clones along with their coupled code fragments so that we can update them consistently in future. However, if such an SPCP clone fragment needs to be refactored, then we should be careful of their relationships across their class boundaries. Overall, 10.27% of all clones in a software system are cross-boundary SPCP clones.

(2) The non-cross-boundary SPCP clones should be considered as the most important candidates for refactoring. Overall, 13.20% of all clones in a software system are non-cross-boundary SPCP clones.

TABLE I. RESEARCH QUESTIONS

SL	Research Question
1	What proportion of the SPCP clones have cross-boundary relationships?
2	Should we discard the cross-boundary SPCP clones from consideration while taking refactoring decisions and also, consider them for tracking?
3	Do cross-boundary SPCP clones have higher change-proneness than the non-cross-boundary SPCP clones?
4	Which types (i.e., Type 1, Type 2, or Type 3) of SPCP clone fragments have higher possibility of having cross-boundary relationships?

(3) The cross-boundary SPCP clones have much higher change-proneness than the non-cross-boundary SPCP clones. The main reason behind this higher change-proneness is that the cross-boundary SPCP clones are generally highly coupled with other code fragments beyond their class boundaries. Thus, such SPCP clones are the places in a software system where we can think of possible restructuring to minimize their coupling.

(4) Considerable proportions of Type 2 and Type 3 SPCP clones have cross-boundary relationships. However, cross-boundary SPCP clones are rare in Type 1 case.

As the non-SPCP clone fragments either evolved independently or rarely changed during evolution, we can exclude them from management considerations. Overall 43% of the SPCP clones of a subject system can have cross-boundary relationships. Our implemented system can automatically identify these and thus, can help us identify the most important candidates for tracking as well as for refactoring.

The rest of the paper is organized as follows. Section II describes the terminology, Section III elaborates on the cross-boundary relationships of SPCP clones, Section IV describes the methodology, Section V answers the research questions based on experimental results, Section VI mentions possible threats to validity, Section VII discusses the related work, and Section VIII 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 [28], 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 change. 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 IV).

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 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, also

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

known as change coupling [6], helps us detect the underlying relationships among program entities in a software system [38]. In our research work we mine evolutionary coupling of SPCP clone fragments to discover their relationships with code fragments beyond their class boundaries.

Association Rule. Evolutionary coupling has been identified by using association rules [1]. 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. 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. [38], *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.

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

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, $X => Y$, determines the probability that Y will change in a commit operation provided that X changed in that commit operation. We determine the confidence of $X => Y$ in the following way.

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

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-C.

III. SPCP Clones having Relationships beyond their Class Boundaries

In this section, we at first discuss SPCP clones and then describe how we detect those SPCP clones each having change coupling with other code fragments (clone and/or non-clone fragments) beyond its class boundary.

A. *SPCP Clones*

The elaboration of SPCP is *Similarity Preserving Change Pattern*. As we defined in our earlier work [24], if two or more clone fragments from the same clone class evolve by receiving only *Similarity Preserving Changes* and/or *Re-synchronizing changes*, then we say that these clone fragments follow a *Similarity Preserving Change Pattern*. We call these clone fragments the *SPCP Clone Fragments* or *SPCP Clones*.

Similarity Preserving Change. Let us consider two code fragments that are clones of each other in a particular revision of a subject system. A commit operation was applied on this revision, and any one or both of these code fragments (i.e., clone fragments) received some changes. However, in the next revision (created because of the commit operation) if these two code fragments are again considered as clones of each other (i.e., the code fragments preserve their similarity), then we say that the code fragments received *Similarity Preserving Change* in the commit operation.

Re-synchronizing Change. Let us consider two code fragments that are clones of each other in a particular revision. A commit operation was applied on this revision, and any one of both of the fragments received some changes in such a way that the code fragments were not considered as clones of each other in the next revision. However, in a later commit operation any one or both of the code fragments received some changes, and because of these changes the code fragments again became clones of each other. Such a converging change followed by a diverging change is termed as a re-synchronizing change.

B. *SPCP Clones having Cross-Boundary Relationships*

In our previous work [24] we mentioned that if two or more clone fragments from a particular clone class are identified as SPCP clones, then they might be important candidates for refactoring, because they evolve by receiving similarity preserving changes or re-synchronizing changes. However, merging these clone fragments into one, that is, removal of all these fragments by a single one is tricky. Here, we should not only think of whether we can merge them but also think about whether we should merge them. If we remove a code fragment from a code-base without being conscious about its relationships with other code fragments, then it is likely that the related code fragments will be negatively affected. These relationships might not be structural. Most of the recent IDEs are capable of pin-pointing the violations or breaking of structural relationships. However, code fragments might have evolutionary coupling relationships, and the IDEs cannot identify if we are going to break such a relationship between two code fragments. By analyzing the evolution history of the clone fragments of our subject systems we observe that an SPCP clone fragment from a particular clone class can also have evolutionary coupling relationships with other two types of code fragments: (1) clone fragments from other clone classes and, (2) non-clone fragments such that the SPCP clone fragment and these other code fragments often need to be changed together (co-changed) correspondingly. In presence of such co-change relationships, also known as change couplings, this SPCP clone fragment should not be removed by refactoring. Removal of this fragment can negatively affect the other related fragments. Thus, this is important to identify SPCP clones with relationships beyond their class boundaries so that we can filter-out them from consideration while taking refactoring decisions and can keep track of them along with their cross-boundary relationships for updating them consistently in future. We detect cross-boundary SPCP clones by analyzing evolutionary coupling. The detection procedure is described below in detail.

C. *Detection of Cross-Boundary SPCP Clones by Mining Evolutionary Coupling*

We at first detect all the SPCP clones from a subject system following the procedure we proposed in our earlier work [24]. We then automatically examine the evolution history of the subject system in order to extract the evolutionary coupling of each SPCP clone with non-clone fragments as well as with clone fragments from other clone classes rather than its own clone class. By examining the evolution history, we determine pairs of co-changed code fragments that can be categorized into the following two categories.

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

(1) ***Different Class Category:*** *Each pair in this category consists of two clone fragments: (1) one is an SPCP clone fragment from a particular clone class, and (2) the other one is a clone fragment (may be an SPCP clone fragment or not) from a different clone class rather than the clone class of the first one such that these two clone fragments co-changed (i.e., changed together) during the past evolution.*

(2) ***Clone Non-clone Category:*** *A pair in this category consists of two code fragments: (1) one is an SPCP clone fragment from a particular clone class, and (2) the other one is a non-clone fragment such that these two code fragments co-changed during the past evolution.*

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. From a particular pair (CF1, CF2) of co-changed code fragments we determine two association rules $CF1 => CF2$ and $CF2 => CF1$ along with their support and confidence values. We consider only those association rules each of which satisfies the following two conditions.

Condition 1. The rule has a support (i.e., the number of times the constituent code fragments co-changed) of at least 2. We discard the lowest support rules (i.e., the rules with support = 1) from consideration because the constituent code fragments in such a rule has a very low probability of having change coupling between them. Such kind of discarding of rules has been done by the previous studies [6], [15].

Condition 2. The rule has a confidence of 1 (i.e., the highest confidence). An association rule *CF1 => CF2* with highest confidence indicates that *each commit operation where CF1 received some changes, CF2 also received some changes*. Thus, it is very much likely that *CF1* and *CF2* have change coupling, and a future change in one fragment will trigger a corresponding change to the other one. The term corresponding change is defined in Section II.

We determine the set of SPCP clone fragments involved in those rules that satisfy the above two conditions. We consider these SPCP clones as the cross-boundary SPCP clones. By excluding these cross-boundary SPCP clones from the whole set of SPCP clones of a subject system, we get the non-cross-boundary SPCP clones for that system.

IV. 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.

We first download all the revisions as noted in Table II for all the subject systems from their open-source SVN repository[2]. Then, for each system we perform nine experimental steps as follows: (1) Method detection and extraction from each of the revisions using CTAGS [3], (2) Detection and extraction of code clones from each revision using the NiCad clone detector [7], (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 [20], (7) Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy, (8) Detection of SPCP clone fragments following technique we proposed in our earlier work [24], and (9) Mining the evolutionary coupling of SPCP clone fragments to identify the cross-boundary SPCP clones. For the details of the first seven steps we refer the interested readers to our earlier work [22]. We have described the final step in Section III-C.

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. We use NiCad [7] for detecting clones because it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [28], [30]. 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 [22]) are considered standard [29].

In this experiment we consider clone and non-clone fragments that reside within methods. 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. If a method contains one or more clone fragments, then we consider the remaining code in the method as a non-clone fragment. If a method does not contain any clone fragment, then we consider the full method as a non-clone fragment. A fully cloned method does not contain a non-clone fragment.

V. Experimental Results and Discussion

We apply our implemented system on each of the six subject systems listed in Table II. Our implemented system automatically determines the SPCP clones, and then identifies those SPCP clones that have relationships across their class boundaries. By analyzing these SPCP clones we answer the research questions mentioned in Table I.

A. RQ 1: What proportion of the SPCP clones of a subject system have cross-boundary relationships?

Answering this research question is important. If it is observed that generally a significant proportion of the SPCP clones of a subject system have cross boundary relationships, then we can consider these clone fragments for tracking along with their cross-boundary relationships in order to update them consistently in future. However, if the number of cross-boundary SPCP clones is too low compared to the total number

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

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

TABLE III. NUMBER OF SPCP CLONE FRAGMENTS

	Ctags	QMail.	Freecol	jEdit	Carol	Jabref
No. of SPCP Clones	229	31	860	902	238	418
% of SPCP Clones w.r.t all clones in the system	43.04	59	55.69	12	29.45	43.63
No. of CBSPCP Clones	116	28	340	442	125	121
% of CBSPCP Clones w.r.t all clones in the system	21.80	53.29	22.02	5.88	15.47	12.63
% of NCBSPCP Clones w.r.t all clones in the system	21.24	5.70	33.67	6.12	13.98	31

SPCP Clones = The clone fragments that evolved following SPCP
No. of CBSPCP Clones = Number of Cross-Boundary SPCP clone fragments
CBSPCP Clones = Cross-boundary SPCP clone fragments
NCBSPCP Clones = Non-cross-boundary SPCP clone fragments

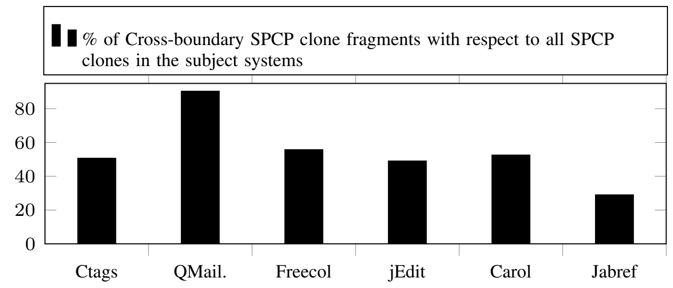


Fig. 2. Percentage of cross-boundary SPCP clone fragments

of SPCP clones in a system, then the detection of such SPCP clones might just be an overhead.

Methodology. We have already mentioned that in our previous study [24] we detected SPCP clone pairs. We merged these pairs to form groups of SPCP clone fragments. While detecting SPCP clone pairs in our previous work, we considered the following two constraints: **(1)** the two constituent SPCP clone fragments in a pair must co-change at least once during evolution, and **(2)** the two SPCP clone fragments in a pair must remain in two different methods. However, in this research work, we did not apply these constraints. Firstly, according to the definition of similarity preserving change pattern (SPCP) [24], two clone fragments from a particular clone class can follow an SPCP without being co-changed at all. So, in this experiment we detect all those SPCP clone pairs where the constituent clone fragments in a particular pair might not co-change at all. Secondly, two clone fragments remaining in the same method can also follow a similarity preserving change pattern and can be important candidates for refactoring. So, in this experiment we consider the same method case too. We show the amount of SPCP clones detected from each of the subject systems in Table III. The proportion of SPCP clones having cross-boundary relationships is shown in Fig. 2.

Fig. 2 shows that *in case of each of the subject systems a considerable amount of the SPCP clones have change couplings with code fragments beyond their class boundaries*. According to our consideration, such SPCP clones should not be considered for removal through refactoring. They should be tracked along with their cross-boundary relationships. Moreover, we have already mentioned that we detect cross-boundary SPCP clones considering the highest confidence rules. As higher confidence indicates stronger change coupling between the constituent code fragments in a rule, we can expect that each of our detected cross-boundary SPCP clones has strong change coupling with non-clone fragments and/or with clone fragments (may be SPCP clone fragments or not) from clone classes other than its own clone class.

For each of our candidate systems we develop two separate XML files containing respectively the cross-boundary and non-cross-boundary SPCP clones from that system. For each cross-boundary SPCP clone fragment, we list the other code fragments (along with their file and starting and ending line numbers) that are related to it. We determine groups of non-cross-boundary SPCP clones following the procedure we proposed in our previous study [24] and include the groups in the XML files. These XML files are available non-line[4].

Answer to RQ 1: In general, *a considerable proportion (overall 43% considering all six subject systems) of the SPCP clones of a subject system have strong change couplings with code fragments beyond their class boundaries. These cross-boundary SPCP clones should be considered as the most important candidates for tracking. We should track them along with their cross-boundary couplings so that we can update them consistently during future evolution. Overall 10.27% of all clones in a software system are cross-boundary SPCP clones. The non-cross-boundary SPCP clones are conservative in the sense that they do not have change couplings with code fragments beyond their class boundaries. Thus, these should be considered as the most important candidates for refactoring. According to our statistics considering all subject systems, overall 13.20% of all clones in a software system are non-cross-boundary SPCP clones.*

B. RQ 2: Should we discard the cross-boundary SPCP clones from consideration while taking refactoring decisions and also, consider them for tracking?

Answering this research question is the central objective of our research work. However, it is difficult to answer this question, because we do not have any empirically established set of characteristics such that a clone fragment that does not have any of the characteristics in the set should not be considered for removal through refactoring. In our previous study we considered that two or more clone fragments from a particular clone class might be important for refactoring only if they evolve following an SPCP (Similarity Preserving Change Pattern). The clone fragments that do not follow SPCP either evolve independently or rarely change during evolution. However, another important characteristic for a clone fragment to be considerable for refactoring is that it is not expected to have change couplings with other code fragments beyond its class boundary. We did not consider this issue in our previous study. If an SPCP clone fragment has such couplings, then it is better not to refactor it, rather it should be tracked along with its cross-boundary couplings so that the maintenance engineers can update them (i.e., the SPCP clone fragment and the code fragments coupled with it beyond its class boundary) consistently in future.

In Fig. 1 (explained in the introduction) we showed an SPCP clone fragment having couplings beyond its class boundary. We also explained that removal of such an SPCP clone fragment might have ripple change effects[1] on the related code fragments remaining outside of its class boundary and also, might negatively affect the future evolution of these related code fragments. Thus, possibly we should not consider refactoring such an SPCP clone fragment. Here we should note

[4]http://goo.gl/r6gDm2

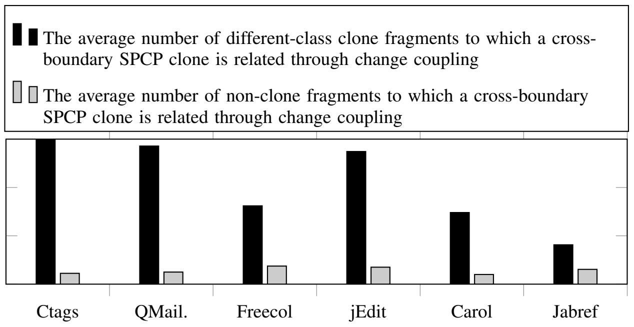


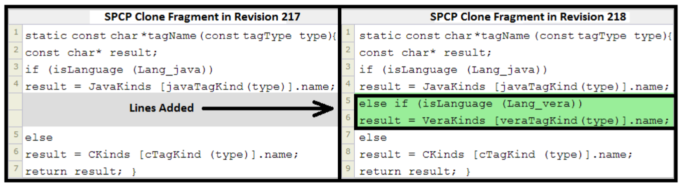
Fig. 3. The average number of code fragments to which a cross-boundary SPCP clone fragment is related beyond its class boundary

that it is impossible to calculate how much negative effect can be caused in future because of the removal of such an SPCP clone fragment. However, we can have an idea from the number of change coupling relationships an SPCP clone fragment currently has. According to our consideration, if a number of fragments are related to a particular code fragment, then removal of that particular fragment might negatively affect the future evolution of all the related code fragments. Thus, by looking at the number of cross-boundary code fragments (i.e., the code fragments beyond class boundary) a particular SPCP clone fragment is related to we can have an idea of how much negative effect might be caused for the removal of that particular SPCP clone fragment.

In this research question, we determine how many change couplings an SPCP clone fragment can have beyond its class boundary. We automatically identify these change couplings by mining association rules as mentioned previously and then manually investigate the association rules to determine whether an SPCP clone fragment is really coupled with code fragments beyond its class boundary. If an SPCP clone fragment has change couplings with code fragments across its class boundary, then we can possibly decide that removal of this clone fragment can negatively affect these related code fragments and thus, can be harmful for future software evolution.

Methodology. We determine all the association rules associating an SPCP clone fragment with other code fragments (non-clone fragments or clone fragments from other clone classes rather than its own class) beyond its class boundary. As we previously mentioned, we consider each of those rules that have the highest confidence (confidence = 1) to ensure the likeliness of change coupling between the code fragments constituting a rule. Considering all the cross-boundary SPCP clone fragments in a subject system, we determine the average number of code fragments (clone fragments from other clone classes, and non-clone fragments) a cross-boundary SPCP clone fragment is associated to by association rules beyond its class boundary. The average numbers for each of the subject system are shown in Fig. 3.

From the figure we see that a cross-boundary SPCP clone fragment can exhibit strong evolutionary coupling (i.e., change coupling) with a considerable number of other code fragments beyond its class boundary. Most of these code fragments are clone fragments from different clone classes rather than its own clone class. We sort the cross-boundary SPCP clone fragments in decreasing order of the number of cross-boundary code fragments they are associated to and then analyze the association rules of the top ten SPCP clone fragments from each subject system. In case of each of the association rules, we


SPCP Clone Fragment in Revision 217
Lines Added

```
static const char *tagName (const tagType type){
const char* result;
if (isLanguage (Lang_java))
result = JavaKinds [javaTagKind (type)].name;
else
result = CKinds [cTagKind (type)].name;
return result; }
```

SPCP Clone Fragment in Revision 218

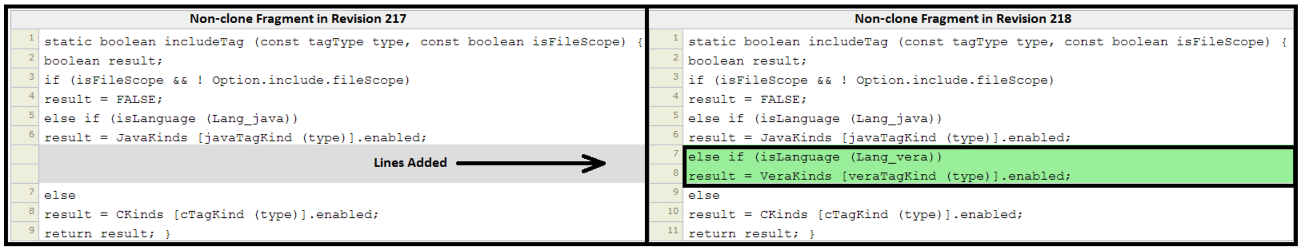
```
static const char *tagName (const tagType type){
const char* result;
if (isLanguage (Lang_java))
result = JavaKinds [javaTagKind (type)].name;
else if (isLanguage (Lang_vera))
result = VeraKinds [veraTagKind (type)].name;
else
result = CKinds [cTagKind (type)].name;
return result; }
```

Fig. 4. Changes occurred to an SPCP clone fragment in the commit operation applied on revision 217 of our subject system Ctags.

determine the commit operations where the constituent code fragments (a cross-boundary SPCP clone fragment and a code fragment beyond its class boundary) co-changed and whether they co-changed correspondingly. According to our manual investigation, each of our investigated cross-boundary SPCP clone fragments was actually related to the code fragments beyond its class boundary. We provide an example of a corresponding change in the following paragraphs.

Example: We provide an example of a corresponding change between an SPCP clone fragment and a non-clone fragment from our subject system Ctags. The SPCP clone fragment is a method called 'tagName' with signature *const char * tagName (const tagType type)*. The non-clone fragment is also a method with name 'includeTag' and signature *boolean includeTag(const tagType type, const boolean isFileScope)*. These two code fragments belong to the same file *c.c*. They co-changed in two commit operations applied on the revisions 217, and 242. Also, both of the association rules constructed from these two code fragments have the highest confidence (confidence = 1). Thus, these two code fragments always co-changed (changed together), and also, there is no commit operation where one fragment changed but the other did not. In such a situation, we can expect strong change coupling between these two code fragments (an SPCP clone fragment, and a non-clone fragment). We manually analyze the changes occurred to these code fragments in both of the commit operations. According to our analysis, the changes occurred to the two code fragments in each of the commit operations were corresponding and thus, the code fragments have change coupling. We show the changes occurred to the two code fragments in the commit operation applied on revision 217 in two figures: Fig. 4, and Fig. 5.

Fig. 4 shows the changes occurred to the SPCP clone fragment (*tagName*), and Fig. 5 shows the changes occurred to the non-clone fragment (*includeTag*). Each figure shows the two instances of the respective code fragment in two revisions 217, and 218. We also highlight the changes between these two instances. From Fig. 4 we see that two lines (statements) were added to the SPCP clone fragment because of the commit on revision 217. If we take a look at Fig. 5, we can see that the same statements were also added to the non-clone fragment in the same commit operation. The changes occurred to the two code fragments imply that those (i.e., the changes) were made focusing on the consistency of the two code fragments. In other words, the SPCP clone fragment and the non-clone fragment co-changed correspondingly in the commit operation. The changes occurred to these two code fragments in the commit 242 are also corresponding according to our manual investigation. Thus, we can expect that these two code fragments are related although there is no caller-callee relationship. According to our analysis, these code fragments have a tendency of co-changing consistently. In such a situation, deletion of the SPCP clone fragment



Non-clone Fragment in Revision 217

```
static boolean includeTag (const tagType type, const boolean isFileScope) {
boolean result;
if (isFileScope && ! Option.include.fileScope)
result = FALSE;
else if (isLanguage (Lang_java))
result = JavaKinds [javaTagKind (type)].enabled;
else
result = CKinds [cTagKind (type)].enabled;
return result; }
```

Non-clone Fragment in Revision 218

```
static boolean includeTag (const tagType type, const boolean isFileScope) {
boolean result;
if (isFileScope && ! Option.include.fileScope)
result = FALSE;
else if (isLanguage (Lang_java))
result = JavaKinds [javaTagKind (type)].enabled;
else if (isLanguage (Lang_vera))
result = VeraKinds [veraTagKind (type)].enabled;
else
result = CKinds [cTagKind (type)].enabled;
return result; }
```

Fig. 5. Changes occurred to a non-clone fragment in the commit operation applied on revision 217 of our subject system Ctags.

('tagName') through refactoring might negatively affect the future evolution of the non-clone fragment. However, if we still want to remove this SPCP clone fragment, then we must take the relationship between this fragment and the non-clone fragment ('includeTag') into consideration so that removal of the SPCP clone fragment does not leave the non-clone fragment in an inconsistent state and also does not affect the future evolution of the non-clone fragment.

Answer to RQ 2. From our discussion and analysis we can say that *the SPCP clone fragments having cross-boundary relationships should not be considered for removal through refactoring. They should be tracked along with their relationships for their consistent updates in future. However, in case we want to remove such a clone fragment, we must consider and analyze its relationships (change couplings in our experiment) with the other code fragments beyond its class boundary so that these other code fragments are not left in an inconsistent state because of the removal of the SPCP clone fragment.*

C. RQ 3: Do cross-boundary SPCP clones have higher change-proneness than the non-cross-boundary SPCP clones?

Motivation. From our answer to the previous research question we understand that a cross-boundary SPCP clone fragment has a tendency of maintaining consistency (i.e., changing consistently) with non-clone fragments as well as with clone fragments from other clone classes. Also, as it is an SPCP clone fragment, it also maintains consistency with other SPCP clone fragment(s) in its own clone class. From this we suspect that possibly cross-boundary SPCP clones have higher change-proneness compared to the non-cross-boundary SPCP clones. Also, our detection mechanism of cross-boundary SPCP clone fragments described in Section III-C ensures that each cross-boundary SPCP clone fragment must co-change at least twice with a non-clone fragment or with a clone fragment from other clone class. In these circumstances it is highly likely that cross-boundary SPCP clones will exhibit higher change-proneness than the non-cross-boundary SPCP clones. However, we investigate this matter in detail in this research question. Such an investigation is important from the perspective of software maintenance.

Literature [19], [20], [23] shows that code clones have higher change-proneness compared to non-cloned code. However, studies [16], [24] also show that not all clone fragments in a software system are highly change-prone, and even some clone fragments never change during software evolution. Thus, this is important to identify which clones are highly change-prone and to investigate the reasons behind their high change-proneness. If our investigation shows that cross-boundary SPCP clones have significantly higher change-proneness than

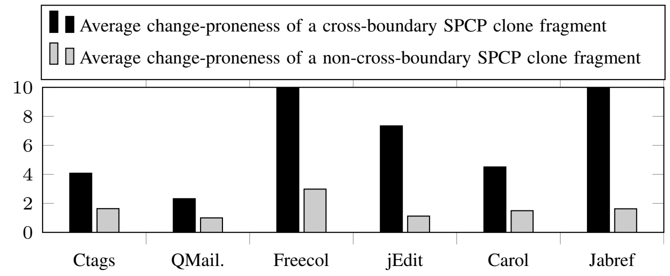


Fig. 6. Comparison of change-proneness between cross-boundary and non-cross-boundary SPCP clone fragments

the non-cross-boundary SPCP clones, then the cross-boundary SPCP clones should be regarded as the hot spots in a software system. In general, highly change-prone code fragments have higher possibilities of introducing bugs and inconsistencies to the software system if the changes occurred to them are not properly propagated to the other related code fragments because of unconsciousness. We know that the cross-boundary SPCP clones are related with many other code fragments beyond their class boundaries. If such an SPCP clone fragment appears to be highly change-prone, then we should be more conscious about the changes occurred to it. If a change occurred to it is not properly propagated to the other related code fragments both inside and outside of its class, then the software system might become inconsistent. Thus, if cross-boundary SPCP clones have high change-proneness, then it is very much important that they should be tracked along with their relationships so that we can change them consistently during future evolution. In this research question we investigate whether cross-boundary SPCP clone fragments have significantly higher change-proneness compared to the non-cross-boundary SPCP clones.

Methodology. To quantify the change-proneness of a particular SPCP clone fragment, we measure the number of times it changed (i.e., the number of commits where it changed) during the whole evolution period of the software system as we did in a previous study [25]. We at first determine all the SPCP clone fragments of a particular subject system, and then separate them into two groups. One group contains the cross-boundary SPCP clone fragments. We call this group the *cross-boundary-group*. The other group contains the non-cross-boundary SPCP clone fragments. We call this group the *non-cross-boundary-group*. We measure the change-proneness of each of the SPCP clone fragments included in each of the two groups. We then determine the average change-proneness per group. These average values are shown in Fig. 6.

Fig. 6 shows that in case of each of the subject systems the change-proneness of the cross-boundary SPCP clones is much higher than the change-proneness of the non-cross-boundary SPCP clones. This is expected according to our previous

TABLE IV. Mann-Whitney-Wilcoxon Tests regarding the significance of difference between change-proneness values of Cross-Boundary-Group and Non-Cross-Boundary-Group

	Ctags	QMail.	Freecol	jEdit	Carol	Jabref
SCBG	116	28	340	442	125	121
SNCBG	113	3	520	460	113	297
p-Value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001

SCBG = No. of Samples (SPCP clone fragments) in Cross-Boundary-Group
SNCBG = No. of Samples in Non-Cross-Boundary-Group
p-Value = Probability Value. If it is less than 0.05 then the difference between the two samples are considered significant.

TABLE V. Cross-Boundary SPCP clones in three clone-types

	Type 1		Type 2		Type 3	
	SPCP	CB-SPCP	SPCP	CB-SPCP	SPCP	CB-SPCP
Ctags	6	0	42	12	133	65
QMailAdmin	0	0	11	9	10	9
Freecol	75	2	143	42	534	164
jEdit	146	66	69	10	458	179
Carol	2	0	48	21	142	58
Jabref	13	0	84	12	236	70

SPCP = Count of SPCP clone fragments
CB-SPCP = Count of cross-boundary SPCP clone fragments

discussion. However, we also wanted to investigate whether the cross-boundary SPCP clones exhibit significantly higher change-proneness than the non-cross-boundary SPCP clones. We perform our investigation in the following way.

In case of this investigation we did not rely on the average change-proneness values, rather we used the actual change-proneness value of each of the SPCP clone fragments in each of the groups. For each of the subject systems, we performed the *Mann-Whitney-Wilcoxon* Tests[5] on the change-proneness values of the SPCP clone fragments of the two groups: *cross-boundary-group*, and *non-cross-boundary-group*. We determine whether the change-proneness values in the *cross-boundary-group* are significantly higher than the change-proneness values in the *non-cross-boundary-group*. For our six subject systems we performed six tests and observed that in case of each test, the difference between the change-proneness values in the two groups was highly significant with *p-value* < 0.001 (for both one-tailed and two-tailed tests). Thus, we can say that the change-proneness values in the *cross-boundary-group* are significantly higher than those in the *non-cross-boundary-group*. The test details are shown in Table IV.

Answer to RQ 3. According to our analysis *cross-boundary SPCP clones have significantly higher change-proneness than the non-cross-boundary SPCP clones. Thus, cross-boundary SPCP clones should be tracked along with their cross-boundary couplings with high priority so that we can change them consistently during future evolution.*

The main reason behind this high change-proneness of the cross-boundary SPCP clones is that each of these has change couplings beyond its class boundary. From Fig. 3 we see that a cross-boundary SPCP clone fragment is generally coupled with a high number of code fragments beyond its class. In other words, these clone fragments are highly coupled. Generally highly coupled code fragments are not desirable in a software system[1], because changes to such a code fragment might cause ripple change effects to the related code fragments. Thus, the cross-boundary SPCP clones are the possible places in a software system where we can think of possible restructuring to minimize their couplings. Also, as the cross-boundary SPCP clones are not suitable for removal through refactoring, we propose efficient tracking of these clone fragments along with their cross-boundary relationships.

D. RQ 4: Which types of SPCP clones have higher possibility of having cross-boundary relationships?

In this research question we wanted to see a comparative scenario of the proportions of SPCP clone fragments as well as of the proportions of cross-boundary SPCP clone fragments in different types of clones of our candidate systems. The findings from this research question can help us understand two important things: **(1)** which type(s) of SPCP clone fragments have lower possibilities of having cross-boundary relationships and thus, are more suitable for refactoring, and **(2)** which type(s) of SPCP clones have higher possibility of having cross-boundary relationships and thus, are more suitable for tracking. We perform our analysis in the following way.

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

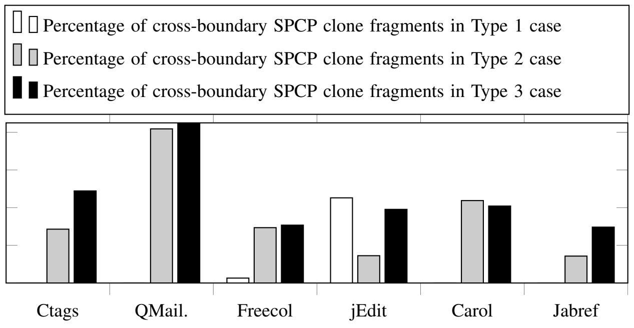


Fig. 7. Comparison of change-proneness between cross-boundary and non-cross-boundary SPCP clone fragments

Methodology. We detect each of the three major types (Type 1, Type 2, and Type 3) of clone fragments separately using the NiCad [7] clone detector and then automatically determine the SPCP clone fragments having cross-boundary change couplings in each clone type. We determine the percentage of such SPCP clone fragments with respect to the total number of SPCP clone fragments considering each clone type. Table V shows the numbers of SPCP clone fragments and cross-boundary SPCP clone fragments in three clone types. The percentages of the cross-boundary SPCP clones are shown in the graph of Fig. 7.

From Table V we see that the number of SPCP clone fragments is generally the highest in Type 3 case among all three types except QMailAdmin. Also, for three subject systems Ctags, Carol, and Jabref we did not get any cross-boundary SPCP clone fragments in Type 1 case. Fig. 7 shows that for most of the subject systems except jEdit and Carol, the percentage of cross-boundary SPCP clones is the highest in Type 3 clones. However, the percentages of cross-boundary SPCP clones are also considerable for the Type 2 cases. In case of jEdit and Carol, the percentages regarding Type 1 and Type 2 cases are the highest ones respectively. Considering all the subject systems it seems that the percentage of cross-boundary SPCP clones is generally the lowest in Type 1 case and the highest in Type 3 case.

Answering RQ 4. According to our investigated subject systems, *the SPCP clones in both Type 2 and Type*

3 cases have high possibilities of having cross-boundary relationships. However, the proportions of cross-boundary SPCP clones are generally very low in Type 1 case.

As Type 1 SPCP clones have lower probabilities of having cross-boundary relationships compared to the SPCP clones in the other two types (Type 2, and Type 3), Type 1 SPCP clones should be considered for refactoring with higher priority. According to our observation, we can even exclude Type 1 clones from consideration while detecting cross-boundary SPCP clones. However, the amounts of cross-boundary SPCP clones in the other two types are considerable. According to our analysis, we should detect cross-boundary SPCP clones considering these two clone types (Type 2, and Type 3) so that we can exclude such SPCP clones from refactoring decision and also, can keep track of them along with their relationships for consistent updates in future.

E. Ranking of SPCP clones for tracking and refactoring

From our previous analysis it is clear that we recommend the cross-boundary SPCP clones as the most important candidates for tracking and the non-cross-boundary SPCP clones as the most important candidates for refactoring. In our previous study we ranked the SPCP clones for refactoring on the basis of their similarity preserving co-changes [24]. According to our findings in this research work, we can apply such a ranking for refactoring the non-cross-boundary SPCP clones. However, the cross-boundary SPCP clones should be treated in a different way. We suggest to rank them on the basis of the number of cross-boundary code fragments they are related to. An SPCP clone fragment having a higher number of cross-boundary relationships can be given a higher rank compared to the other SPCP clone fragments having comparatively a lower number of cross-boundary relationships. In general, the more a code fragment is coupled, the more it is challenging to be changed. Tracking of such a code fragment with all of its couplings can help us change it consistently in future. Thus, we believe that our consideration regarding the ranking of cross-boundary SPCP clones for tracking is reasonable. We rank the cross-boundary SPCP clones detected from each of our candidate systems considering our proposed ranking mechanism. The XML files containing these ranked cross-boundary SPCP clones are available on-line[4].

VI. Threats to the Validity

We used the NiCad clone detector [7] for detecting clones. For different settings of NiCad, the statistics that we present in this paper might be different. Wang et al. [36] 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 [29] and with these settings NiCad can detect clones with high precision and recall [28], [30].

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the cross-boundary relationships of SPCP clones. However, our candidate systems were of 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 by minimizing clone refactoring effort and suggesting important candidates for tracking.

VII. Related Work

Numerous studies have been conducted regarding the detection, impact analysis [9], [10], [13], [16]–[20], management [35], refactoring [3], [14], [33], [37] and tracking [8], [12], [21], [34] of code clones.

A number of refactoring approaches [3], [14] select clones for refactoring on the basis of the abstract syntax tree representation of the code base. Higo et al. [11] selected clones for refactoring (implementing a tool called CCShaper) based on the lexical analysis of the source code. Bouktif et al. [4] proposed an optimal schedule for refactoring clones using genetic algorithm. Zibran and Roy [37] proposed a conflict aware optimal scheduling algorithm for clone refactoring using constraint programming. Tairas and Gray [33] developed an Eclipse plug-in, CeDAR, for the purpose of clone refactoring.

A number of techniques for clone tracking also exist. Duala-Ekoko and Robillard [8] implemented an Eclipse plug-in 'CloneTracker' for tracking clones. Jablonski and Hou [12] developed a tool called CReN to track copy-paste activities. Miller and Myer [21] proposed a technique for simultaneous editing of multiple clone fragments. Toomin et al. [34] developed a clone tracking tool called *Codelink*.

We see that a number of studies and techniques for clone refactoring and tracking already exist. These techniques and tools consider all clones in a software system for refactoring and tracking. However, our study in this paper is different and unique in the sense that we detect only those clones that are important for refactoring or tracking.

Previously we conducted a study [24] to identify important clones for refactoring. We defined a particular clone change pattern called SPCP (*Similarity Preserving Change Pattern*) and proposed a mechanism for detecting clones that follow SPCP. The non-SPCP clones are not important for refactoring or tracking, because they either evolve independently or rarely change during evolution. We proposed that the SPCP clones could be important candidates for refactoring. However, an SPCP clone fragment can have couplings with other code fragments beyond its class boundary. Such cross-boundary SPCP clones should not be considered for removal through refactoring. They should be considered as important candidates for tracking. In this research work we propose a mechanism for automatically identifying cross-boundary SPCP clones. We also propose a particular ranking mechanism for prioritizing the cross-boundary SPCP clones for tracking. We perform an in-depth empirical study on both the cross-boundary and non-cross-boundary SPCP clones. From our empirical evaluation we suggest that cross-boundary SPCP clones are the most important candidates for tracking and the non-cross-boundary SPCP clones are the most important candidates for refactoring.

In a previous study [26] we analyzed the evolutionary couplings of clone fragments in order to predict their future co-change candidates. However, our study in this paper is different in the sense that here we focus on identifying clones that are important for tracking or refactoring. Our implemented system in this research work can automatically extract the important clones for refactoring (non-cross-boundary SPCP clones) as well as for tracking (cross-boundary SPCP clones). Thus, our study is important for better management of code clones.

VIII. Conclusion

In this paper we perform an in-depth empirical study on the identification of clone fragments that are important for refactoring or tracking. We at first detect all the SPCP clones (i.e., the clone fragments that evolved following a similarity preserving change pattern) in a software system. We analyze the evolutionary coupling of the SPCP clones and identify those SPCP clones that have change couplings (i.e., evolutionary couplings) with other code fragments beyond their class boundaries. According to our consideration, these cross-boundary SPCP clones should not be considered for removal through refactoring, because removal of such clone fragments might negatively affect the future evolution of the related code fragments beyond the class boundaries. We consider these as the most important candidates for tracking. We suggest the non-cross-boundary SPCP clones to be the most important candidates for refactoring. Our implemented prototype tool can automatically identify both cross-boundary and non-cross-boundary SPCP clones by analyzing software evolution history.

We apply our prototype tool on six diverse subject systems written in two programming languages and detect the cross-boundary and non-cross-boundary SPCP clones. According to our empirical study involving rigorous manual analysis, overall 43% of the SPCP clone fragments have cross-boundary relationships. Cross-boundary SPCP clones exhibit significantly higher change-proneness than the non-cross-boundary SPCP clones. The reason behind this higher change-proneness is that cross-boundary SPCP clones are generally highly coupled. As lower coupling is always desirable in software systems, cross-boundary SPCP clones are possible places in a software system where we can think of possible restructuring to minimize their coupling. We also observe that the percentage of cross-boundary SPCP clones is generally the lowest in Type 1 case, and the highest in Type 3 case. We believe that automatic detection of cross-boundary as well as non-cross-boundary SPCP clones will help us in better management of code clones in terms of both tracking and refactoring.

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, M. D. Penta, "How clones are maintained: An empirical study", Proc. *CSMR*, 2007, pp. 81-90.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring", Proc. *WCRE*, 2000, pp. 98 – 107.

[4] S. Bouktif, G. Antoniol, E. Merlo, M. Neteler, "A Novel Approach to Optimize Clone Refactoring Activity", Proc. *GECCO*, 2006, pp.1885 – 1892.

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

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

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

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

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

[11] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, "Refactoring support based on code clone analysis", *LNCS*, 2004, 3009: 220 – 233.

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

[14] N. Juillerat, B. Hirsbrunner, "An algorithm for detecting and removing clones in Java code", *SeTra*, 2006.

[15] 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.

[16] M. Kim, V. Sazawal, D. Notkin, G. Murphy, "An empirical study on code clone genealogies", Proc. *FSE*, 2005, pp. 187 – 196.

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

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

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

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

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

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

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

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

[26] M. Mondal, C. K. Roy, K. Schneider, "A Fine-Grained Analysis on the Evolutionary Coupling of Cloned Code", Proc. *ICSME*, 2014, 10 pp. (to appear).

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

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

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

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

[31] C. K. Roy, M. F. Zibran, R. Koschke,"The vision of software clone management: Past, present, and future (Keynote paper)", Proc. *CSMR-WCRE*, 2014, pp. 18 – 33.

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

[33] R. Tairas, J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities", *Information and Software Technology*, 2012, 54(12):1297 – 1307.

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

[35] R. Venkatasubramanyam, S. Gupta, H. K. Singh, "Prioritizing Code Clone Detection Results for Clone Management", Proc. *IWSC*, 2013, pp. 30 – 36.

[36] T. Wang, M. Harman, Y. Jia, J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation", Proc. *ESEC/FSE*, 2013, pp. 455 – 465.

[37] M. F. Zibran, C. K. Roy, "Conflict Aware Optimal Scheduling of Prioritised code clone refactoring", *IET Software*, 2013, pp. 167 – 186.

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