# Automatic Ranking of Clones for Refactoring through Mining Association Rules

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

***Abstract*—In this paper, we present an in-depth empirical study on identifying clone fragments that can be important refactoring candidates. We mine association rules among clones in order to detect clone fragments that belong to the same clone class and have a tendency of changing together during software evolution. The idea is that if two or more clone fragments from the same class often change together (i.e., are likely to co-change) preserving their similarity, they might be important candidates for refactoring. Merging such clones into one (if possible) can potentially decrease future clone maintenance effort.**

**We define a particular clone change pattern, the Similarity Preserving Change Pattern (SPCP), and consider the cloned fragments that changed according to this pattern (i.e., the SPCP clones) as important candidates for refactoring. For the purpose of our study, we implement a prototype tool called MARC that identifies SPCP clones and mines association rules among these. The rules as well as the SPCP clones are ranked for refactoring on the basis of their change-proneness. We applied MARC on thirteen subject systems and retrieved the refactoring candidates for three types of clones (Type 1, Type 2, and Type 3) separately. Our experimental results show that SPCP clones can be considered important candidates for refactoring. Clones that do not follow SPCP either evolve independently or are rarely changed. By considering SPCP clones for refactoring we not only can minimize refactoring effort considerably but also can reduce the possibility of delayed synchronizations among clones and thus, can minimize inconsistencies in software systems.**

***Keywords*-Clone Change Pattern; Association Rule; Clone Class; Clone Refactoring;**

## I. Introduction

Code cloning is a common yet controversial practice frequently employed by programmers during both development and maintenance of software systems. Cloning involves copying a code fragment from one place and pasting it in one or more additional places in the code-base with or without modifications causing the same or similar code fragments to be scattered throughout the system. The original code fragment (i.e., the code fragment from which the copies were created) and the pasted code fragments are clones of one another.

Numerous empirical studies [8], [10], [13], [14], [15], [24] have been conducted identifying the possible impact of clones on software maintenance. While a number of studies [13], [14], [8] reported that clones are beneficial to both software development and maintenance, there is strong empirical evidence [15], [10] of the negative effects of clones, including hidden bug propagation and unintentional inconsistent changes. Also, higher number of clones indicate higher change-proneness of the software systems [18] as well as higher maintenance effort and cost. The negative effects of clones indicate the necessity of clone refactoring. Clone refactoring refers to the task of merging several clone fragments (that are similar to one another) into a single one (if possible).

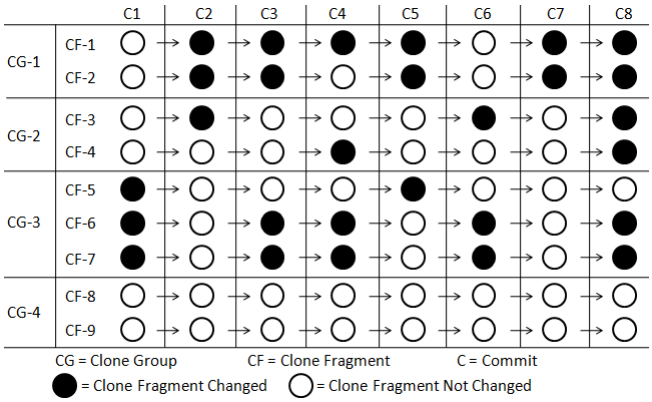


Fig. 1. Change history of clone fragments

**Motivation.** A number of clone refactoring techniques [2], [11], [22], [29], [7], [23] have been proposed by different studies. Before refactoring, it is important to identify the clones that are our primary refactoring candidates because, there can be a large number of clones in a system and not all of them need to be refactored [12]. That is, we should identify clones that would be important to refactor. Clone refactoring may not be able to be fully automated as it may require critical analysis by the programmer. Thus, a significant amount of effort and cost might sometimes need to be spent for refactoring clones. Identifying and ranking clones according to refactoring need can help us minimize refactoring effort and cost, because we are able to focus on just those clones that are important to be refactored, and we can leave the clones where refactoring is less important or unnecessary. However, there is no existing study that focuses on how we should identify and rank the important clones (from the whole set of clones in a code-base) for refactoring. Zibran and Roy [29] proposed an optimal scheduling algorithm for clone refactoring considering all the clones in the code-base. In our study, we propose to identify only important clones for refactoring based on their clone evolution history. The scheduling algorithm [29] can then be applied only on these important clones keeping the rest out of the desk.

According to our consideration, all clones in a software system might not need to be refactored with equal importance. We use the clone change example presented in Fig.1 to explain this. We see that there are nine code fragments forming

four groups (CG-1 through CG-4). The code fragments in a particular group are clones of one another. The change history (in Fig. 1) of these code fragments implies the followings.

**(1)** The clone fragments, CF-1 and CF-2 in group CG-1, are likely to be related to each other and they have received corresponding changes. In other words, it is likely that the changes were made to these clones focusing on their consistency. The same is also true for the two code fragments CF-6 and CF-7 in CG-3.

**(2)** It is likely that the clone fragments, CF-3 and CF-4 in group CG-2, have experienced independent evolution. Because of the independent evolutions, CF-3 and CF-4 might not be regarded as clones of each other after a particular evolution period. Out of the three clone fragments in group CG-3, clone fragments CF-6 and CF-7 seem to be related as they received corresponding changes. However, the other clone fragment CF-5 seems to have a tendency of independent evolution.

**(3)** The clone fragments, CF-8 and CF-9, are not change-prone and it is likely that in future they will exhibit lower change-proneness compared to the others. Thus, these clones are not likely to add change effort during maintenance.

The example in Fig.1 implies that refactoring clone fragments might not always be appropriate, since clones might evolve independently without preserving similarity. Also, if some clone fragments do not change during evolution we might not consider refactoring those clone fragments, since they do not require additional change effort during software evolution. Thus, when clone fragments belonging to the same clone class change consistently during evolution these clone fragments might be important refactoring candidates. Moreover, a clone class may contain $n$ clone fragments, however, if it is observed that only a subset of the clone fragments change consistently while others are either evolving independently or are rarely changing we should consider refactoring only those consistently changing clone fragments. If the consistently changing clones can be merged through refactoring we can reduce the effort spent for changing clones. The co-change histories of the clones: (1) CF-1 and CF-2 in CG-1, and (2) CF-6 and CF-7 in CG-3 indicate that such clone fragments can be identified by mining association rules among clones.

**Contribution.** Focusing on the above discussion we define a particular clone change pattern called ***SPCP (Similarity Preserving Change Pattern)*** such that the clone fragments that change following this pattern (i.e., SPCP clones) are likely to be important candidates for refactoring. We describe SPCP in Section IV. For the purpose of our study, we develop a prototype tool **MARC** (Mining Association Rules among Clones) that can detect all SPCP clones from a subject system and then mines association rules among SPCP clones. **MARC** ranks these rules according to their support and confidence values (defined in Section III). According to our investigation on the rules as well as SPCP clones retrieved by **MARC**,

**(1)** *SPCP clones can be important candidates for refactoring. The clones that do not follow SPCP either evolve independently or are rarely changed during evolution. Thus, we can mainly focus on the SPCP clones for refactoring.*

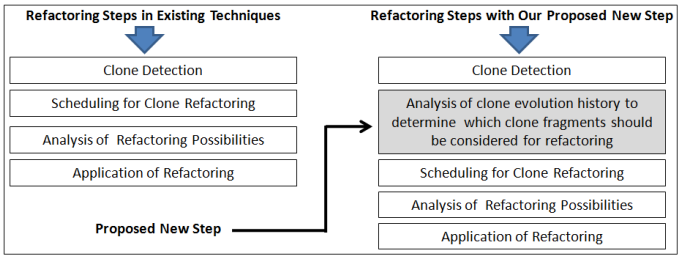


Fig. 2. Proposed refactoring step

**(2)** *Overall, only 7.04% of the clones existing in a code-base are SPCP clones. Also, for each of 63.44% of the association rules retrieved by MARC, the corresponding SPCP clones are method clones and also, they belong to the same source code file. Thus, automatic identification and ranking of SPCP clones can save a considerable amount of time, effort and cost for clone refactoring, because we could leave the remaining 92.96% of the clones in the code-base without refactoring.*

**(3)** *From our manual analysis on 224 association rules from all 13 subject systems (considering the top 10 rules of each clone-type), for overall 64.37% of the association rules we can suggest a standard refactoring technique*[1].

**(4)** *A considerable amount (overall 11.64%) of SPCP clones can receive resynchronizing changes. Thus, refactoring of SPCP clones can minimize delayed synchronizations among clone fragments and can minimize unwanted inconsistencies in software systems.*

The rest of the paper is organized as follows: Section II describes the significance of our study, Section III describes the terminology, Section IV elaborates on the SPCP (similarity preserving change pattern), Section V presents experimental results and discussion, Section VI mentions some threats to validity, Section VII discusses related work, and Section VIII concludes the paper mentioning future work.

## II. Significance of our Study

Existing clone refactoring studies and techniques mainly consider four refactoring steps (cf. Fig. 2): (1) detection of clones, (2) scheduling of clones for refactoring (3) analysis of refactoring possibilities on the basis of different cloning situations, and (4) application of selected refactoring. Our study differs in that we propose an additional step after clone detection that involves the analysis of the clone evolution history to determine the clones we should consider as the primary refactoring candidates. According to our expectation, this additional step will minimize the clone refactoring effort and task considerably, because this step filters out a significant portion of clones from the refactoring target list based on their change pattern and change-proneness. According to our analysis the excluded clones do not need to be refactored with the same importance as the ones included in the target list, since they either changed rarely or changed independently.

## III. Terminology

**Types of Clones.** We conducted our experiment considering exact (Type 1) and near-miss clones (Type 2 and Type 3

[1]Refactoring Techniques: http://www.refactoring.com/catalog/

clones). If two code fragments are exactly the same disregarding comments and indentations, they are Type 1 clones of each other. 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.

**Clone Class.** A group (i.e., two or more) of clone fragments that are the same (Type 1) or similar (Type 2 or Type 3) to one another form a clone class.

**Cloned Method.** If a method contains cloned (Type 1, Type 2, or Type 3) lines, we call this method a cloned method. If all lines of a method are cloned lines, then this method is a fully cloned method. If a method contains both cloned and non-cloned lines, we call this method a partially cloned method.

**Method Clones.** If two or more methods are clones (Type 1, Type 2, or Type 3) of one another, we refer to these as method clones. Method clones are fully cloned methods.

**Association Rule.** An association rule [1] is an expression of the form $X$ => $Y$ where $X$ is the antecedent and $Y$ is the consequent. Each of $X$ and $Y$ is a set of one or more program entities. The meaning of such a rule in our context is that if $X$ gets changed in a particular commit operation, $Y$ also has the tendency of getting changed in that commit operation. We can determine the *confidence* or strength of a particular association rule by determining the *support* of its constituent parts.

**Support and Confidence.** Support is the number of commit operations in which an entity or a group (two or more) of entities changed together. We 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 changed in four commits: 2, 5, 6, and 10. E2 changed in six commits: 4, 6, 7, 8, 10, and 13. Thus, *support(E1) = 4* and *support(E2) = 6*. However, *support(E1, E2) = 2*, because E1 and E2 changed together in two commits: 6, and 10. Also, *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.

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

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*. In our experiment we consider those association rules where each of $X$ and $Y$ consist of a single clone fragment from the same clone class. Such a rule can be expressed as $x$ => $y$ where $x$ and $y$ are two different clone fragments belonging to the same clone class.

## IV. Similarity Preserving Change Pattern

Our prototype tool, MARC, mines association rules considering those clone fragments that followed a *similarity preserving change pattern (SPCP)* during evolution. It can also rank these rules on the basis of change-proneness (described in Section IV-D) of the participating clones. We define *similarity preserving change pattern* in the following way.

### A. Definition of Similarity Preserving Change Pattern

If two clone blocks received either ***similarity preserving changes*** or ***re-synchronizing changes*** or both during evolution, then we say that these clone blocks follow a similarity preserving change pattern (i.e., are SPCP clones).

**Definition of Similarity Preserving Change.** We consider two clone blocks $CB1$ and $CB2$ that belong to the same clone class, say $CLS1$, in revision $R_i$. Suppose, a commit operation $C_i$ on $R_i$ changes any (one or both) of these clone fragments. If in revision $R_{i+1}$ (created because of commit $C_i$) these two clone blocks, $CB1$ and $CB2$, again remain in one particular clone class (which might not be $CLS1$), then we say that $CB1$ and $CB2$ have received a ***similarity preserving change*** in commit operation $C_i$. If both of the clone blocks (i.e., $CB1$ and $CB2$) change preserving their similarity in such a commit then we call this change a ***similarity preserving co-change (SPCO)***. If the SPCP of two clone blocks contains SPCOs, then it is likely that the participating clone blocks have changed consistently (evaluated in Section V-A).

**Definition of Re-synchronizing Change.** Suppose, two clone blocks $CB1$ and $CB2$ belong to the same clone class, $CLS1$, in revision $R_i$. The commit $C_i$ on revision $R_i$ modified any of these clone blocks in such a way that $CB1$ and $CB2$ could not be considered as clones of each other in revision $R_{i+1}$ (i.e., $CB1$ and $CB2$ may diverge into two different clone classes or one or both of these might not be regarded as a clone fragment). However, in a later commit operation, say $C_{i+n}$ where $n$ >= 1, any one or both of $CB1$ and $CB2$ changed in such a way that $CB1$ and $CB2$ become clones of each other (they converged into one clone class) again. Such a converging change followed by a diverging change is termed as a ***re-synchronizing change*** in our experiment.

### B. Example of a Similarity Preserving Change Pattern

Fig. 3 contains an SPCP followed by the clone blocks, $CB1$ and $CB2$. As indicated in the figure, these clone blocks belong to two methods $M1$ and $M2$ respectively. We see that in each of the commit operations, C2 to C7, the clone blocks $CB1$ and $CB2$ received *similarity preserving change*. In commit C1, none of the clone blocks were changed. We now consider the commit operation C2. Before this commit both of the clone blocks (CB1 and CB2) belonged to the clone class CLS1. After this commit, the clone blocks belonged to CLS2. Although CLS1 and CLS2 are different clone classes, we see that before or after this commit operation both clone blocks belonged to a single clone class. Thus, CB1 and CB2 preserved their similarity after commit C2. In other words, CB1 and CB2 received a *similarity preserving change* in commit C2. We also see that these two clone blocks received *similarity preserving change*s in each of the commits C3 to C7. Moreover, the similarity preserving changes in commits C2, C5, C6, and C7, are SPCOs (similarity preserving co-changes). Finally, the changes in commits C8 to C10 can be considered as an

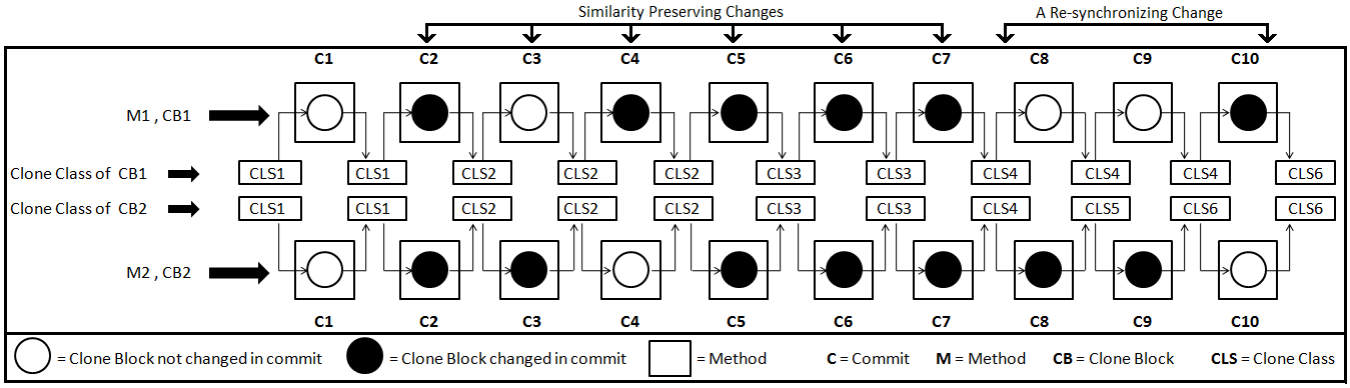


Fig. 3. Similarity Preserving Change Pattern (SPCP)

example of re-synchronizing change. Because of the change in C8, $CB2$ diverged into a different clone class, CLS5, and thus, $CB1$ and $CB2$ could not be considered as clones of each other. However, the change in commit C10 is a converging change because, both of the clone blocks again converged into a single clone class after this commit.

### C. Mining Association Rules Considering SPCP

In this experiment, we consider clones residing in methods. Thus, both fully cloned and partially cloned methods have been investigated. For a particular subject system, we collect all of its revisions (mentioned in Table I), then extract the methods in each revision using CTAGS, and then determine method genealogies following the technique proposed by Lozano and Wermelinger [15]. We detect clones in each revision using NiCad [5] and then map the clones to the already detected methods of the corresponding revisions. As method genealogies are already detected, after clone mapping we can easily track the evolution of each clone fragment. Finally, we detect changes between every two consecutive revisions and map these changes to the methods as well as clones located inside the methods. For the details of these steps and NiCad setup we refer the readers to our earlier work [17].

**Extraction of Association Rules.** After the preliminary steps we determine all possible pairs of clones. A possible pair of clones consists of two clone fragments $CF1$ and $CF2$ from the same clone class such that they together followed an *SPCP* during the evolution and are alive in the last revision. As we map clones to methods, $CF1$ and $CF2$ reside in methods. From such a clone pair we can determine two association rules: $CF1$ => $CF2$, and $CF2$ => $CF1$. According to the definition, these two rules have the same support value. However, their confidences can be different. We determine support by the number of *similarity preserving co-changes (SPCOs)* in the SPCP followed by the clone fragments in a rule. We extract all association rules with a minimum support of 1.

### D. Ranking of Association Rules

According to our consideration and discussion in the introduction, if a rule detected by our prototype tool has a higher support value (i.e., higher SPCO count) compared to the others, we should assume a higher priority for refactoring the associated clone fragments. Our decision of ranking considering the support value is reasonable from two perspectives.

**(1)** Higher support value (i.e., higher SPCO count) for a rule (consisting of two SPCP clone fragments) indicates a higher likelihood that the participating clone fragments have changed consistently during evolution. The underlying assumption is that *in a similarity preserving co-change (i.e., in a SPCO), the two participating SPCP clone fragments generally change consistently.* We empirically evaluate this in Section V-A. Intuitively, if two clone fragments have a tendency of changing consistently, then changes in one fragment in a particular commit generally require corresponding changes to the other one in that commit.

**(2)** Higher support value for a rule is an evidence that the corresponding SPCP clones have exhibited higher change-proneness (i.e., changed in higher number of commits) during the past evolution compared to the other SPCP clones in other rules. Thus, considering the existing evolution history we rank the association rules according to the decreasing order of change-proneness of the corresponding SPCP clones. As it is difficult to be certain about the future change-proneness of the rules as well as SPCP clones, our decision of ranking relying on the past history is reasonable. We assume higher ranks for those rules as well as SPCP clones that exhibited higher change-proneness in the past. However, higher support might also be an indicator of higher change-proneness of the associated SPCP clones in future. We do not investigate this issue (i.e., future change-proneness) in this research work.

### E. Finding Groups of Refactoring Candidates

Each rule consists of two SPCP clones. This is also possible that more than two clone fragments from the same class are preserving their similarity during evolution following a similarity preserving change pattern (SPCP). Thus, this is convenient to determine groups of refactoring candidates from these rules where a group contains two or more clones (from the same clone class) and all the clones in a group have evolved following a SPCP. Suppose two clone blocks, CB1 and CB2, formed a rule because they followed a SPCP. If there is another rule consisting of the clone blocks CB2 and CB3 (CB2 is common in these two rules), then we can form a group consisting of CB1, CB2, and CB3, because according to the conditions in Section IV-A these three clone blocks together

| SL | Clone Blocks That Followed SPCP | Rule Details |
|---|---|---|
| 1 | **Clone Block 1 (CB1)**<br>File: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java<br>Starting line = 57, Ending line = 69 (revision = 156)<br>Cloned Method Name = bind<br>**Clone Block 2 (CB2)**<br>File: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java<br>Starting line = 199, Ending line = 211 (revision = 156)<br>Cloned Method Name = destroySubcontext | Support / SPCO Count = 7<br>Confidence of Rule (CB1 => CB2) = 1.0<br>Confidence of Rule (CB2 => CB1) = 1.0<br>Commits with SPCOs = 42 51 54 58 91 153 156<br>Methods are fully cloned.<br>Clones are from the same file. |
| 2 | **Clone Block 1 (CB1)**<br>File: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java<br>Startingline = 135, Ending line = 147 (revision = 156)<br>Cloned Method Name = rename<br>**Clone Block 2 (CB2)**<br>File: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java<br>Starting line = 109, Ending line = 121 (revision = 156)<br>Cloned Method Name = unbind | Support / SPCO Count = 7<br>Confidence of Rule (CB1 => CB2) = 1.0<br>Confidence of Rule (CB2 => CB1) = 1.0<br>Commits with SPCOs = 42 51 54 58 91 153 156<br>Methods are fully cloned.<br>Clones are from the same file. |

Fig. 4. Rules of Type 3 clones sorted in decreasing order of support values

followed a SPCP. Focusing on this fact, MARC determines groups of refactoring candidates. We term each group as a SPCP clone-class. In Table II we report the count of groups for each subject system considering each clone-type.

### *F. Tool Support*

We have developed our prototype tool such that for a particular clone-type of a particular subject system, it generates two XML files. One file contains the ranked rules and the other one contains the SPCP clone groups merging these rules. While each rule in the first file contains only two SPCP clones, a group in the second file may contain more than two SPCP clones. For each SPCP clone (whether in a rule or in a group) we include the starting and ending line numbers of the clone fragment, the name of the method containing the clone fragment, and the starting and ending line numbers of this method considering the last (i.e., the latest) revision so that we can easily trace the clone fragment for refactoring in the latest revision of the candidate software system. Our prototype tool is now available [16] with instructions on how to use it. The six XML files containing the association rules and groups of three types of SPCP clones of our subject system Freecol are also available on-line [16].

In order to assist in analyzing the evolution of SPCP clones, our tool also shows important information regarding the ranked rules including (i) the list of commits where the SPCOs occurred, (ii) whether the participating clone blocks are method clones, (iii) whether the clones belong to the same file, (iv) the support and confidence values, (v) the file paths and starting and ending line numbers of the clone blocks and container methods in the revision where the last SPCO occurred. Such a ranking with all information regarding the SPCP clones is presented in Fig. 4 that shows the top two rules of total 481 rules retrieved for Type 3 case of Carol.

## V. Experimental Results and Discussion

We applied our prototype tool MARC on each of the thirteen subject systems listed in Table I and detected all SPCP clones and association rules & groups formed by these SPCP clones considering three clone-types (Type 1, Type 2, and Type 3) separately. We manually examined the changes occurred to the SPCP clones. According to our observation, implementation of the changes (*similarity preserving change*s and *resynchronizing change*s) required proper analysis by the responsible programmers. The changes were not tool generated

TABLE I
SUBJECT SYSTEMS

| | System | Domain | LOC | Revisions |
|---|---|---|---|---|
| **Java** | freecol | Game | 91,626 | 1950 |
| | jEdit | Text Editor | 1,91,804 | 4000 |
| | Plandora | Project Management | 94,076 | 73 |
| | Carol | Game | 25,092 | 1699 |
| | OpenYMSG | Yahoo Messenger | 15,553 | 297 |
| **C** | Ctags | Code Def. Generator | 33,270 | 774 |
| | QMail Admin | Mail Management | 4,054 | 317 |
| | GNUMake | Auto-build System for C/C++ Projects | 68,095 | 863 |
| | Uniproc | | | |
| **C#** | MonoOSC | Formats & Protocols | 14,883 | 355 |
| | GreenShot | Multimedia | 37,628 | 999 |
| **Python** | Pyevolve | Artificial Intelligence | 8.809 | 200 |
| | Ocemp | Game | 57,098 | 438 |
| | Noora | Development Tool | 14,862 | 140 |

and thus, these were not just reformatting or code transformations (e.g., replacement of for loops by for-each loops). In the following subsections and also in Fig. 6 we have mentioned and explained such changes. By analyzing our experimental results we answer the following five research questions.

RQ 1. Can we minimize clone refactoring effort and cost by considering SPCP clones for refactoring?
RQ 2. Are the changes occurring to the clone fragments in SPCOs (similarity preserving co-changes) consistency ensuring changes?
RQ 3. Can SPCP clones be candidates for refactoring?
RQ 4. What are the characteristics of the clones that change following similarity preserving change pattern (SPCP)?
RQ 5. What ratio of the SPCP clones do receive resynchronizing changes?

**Statistics of the important refactoring candidates:** Table II shows the statistics of the clone fragments that followed SPCP. We regard these cloned fragments as the important refactoring candidates. For each type of clones of a particular subject system we determine the followings considering all revisions: **(1)** Number of clone classes per revision (**CC**), **(2)** Number of clone fragments per revision (**CF**), **(3)** Number of SPCP clones that is, the number of clone fragments that followed *similarity preserving change pattern* during evolution (**SCF**), **(4)** Number of SPCP clone-classes (**SCC**), **(5)** The percentage of SPCP clones (**PSCF**), and **(6)** Number of association rules formed by the SPCP clones (**CR**). These six measures for each clone type are presented in Table II. Looking at the percentages (**PSCF**) of the cloned fragments that followed SPCP (i.e., the important candidates for refactoring) we realize that *the number of important refactoring candidates can be considerably smaller compared to the total number of clone fragments in a system.*

From Table II, for each clone type, we also determined the overall percentages of the SPCP clones (denoted by *Overall-PSCF*) considering all subject systems according to the following equation.

$$\textit{Overall-PSCF}_{Ti} = \frac{\sum_{for\ all\ systems} SCF_{T1}}{\sum_{for\ all\ systems} CF_{T1}} \quad (2)$$

Here, $Ti$ (i = 1, 2, or 3) denotes a particular clone-type (Type 1, Type 2, or Type 3), $SCF_{Ti}$ denotes the number

TABLE II
STATISTICS REGARDING THE IMPORTANT CLONED FRAGMENTS (THE SPCP CLONES) RETRIEVED BY MARC

| | | Type 1 | | | | | | Type 2 | | | | | | Type 3 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **System** | **CC** | **CF** | **SCC** | **SCF** | **PSCF** | **CR** | **CC** | **CF** | **SCC** | **SCF** | **PSCF** | **CR** | **CC** | **CF** | **SCC** | **SCF** | **PSCF** | **CR** |
| **Java** | Freecol | 93 | 251 | 27 | 56 | 22.31 | 31 | 95 | 259 | 28 | 63 | 24.32 | 49 | 255 | 799 | 78 | 180 | 22.52 | 147 |
| | jEdit | 1516 | 4284 | 3 | 8 | 0.18 | 7 | 115 | 531 | 27 | 62 | 11.67 | 46 | 399 | 2009 | 51 | 108 | 5.37 | 64 |
| | Plandora | 77 | 297 | 5 | 10 | 3.36 | 5 | 160 | 633 | 2 | 4 | 0.63 | 2 | 381 | 1748 | 19 | 48 | 2.74 | 44 |
| | Carol | 31 | 154 | 15 | 51 | 33.11 | 39 | 30 | 185 | 17 | 58 | 31.35 | 80 | 69 | 300 | 22 | 127 | 42.33 | 481 |
| | OpenYMSG | 10 | 24 | 3 | 8 | 33.33 | 7 | 10 | 22 | 0 | 0 | 0.0 | 0 | 40 | 112 | 7 | 16 | 14.28 | 10 |
| **C** | Ctags | 11 | 29 | 3 | 6 | 20.68 | 3 | 19 | 46 | 4 | 8 | 17.39 | 4 | 56 | 168 | 15 | 33 | 19.64 | 21 |
| | QMailAdmin | 15 | 49 | 3 | 6 | 12.24 | 3 | 11 | 43 | 1 | 2 | 4.65 | 1 | 13 | 77 | 4 | 8 | 10.38 | 4 |
| | GNUMake Uniproc | 144 | 308 | 3 | 6 | 1.94 | 3 | 28 | 69 | 0 | 0 | 0.0 | 0 | 57 | 199 | 2 | 5 | 2.51 | 3 |
| **C#** | MonoOSC | 4 | 21 | 3 | 6 | 28.57 | 3 | 3 | 6 | 1 | 2 | 33.33 | 1 | 9 | 23 | 8 | 17 | 73.91 | 10 |
| | GreenShot | 168 | 379 | 2 | 4 | 1.05 | 2 | 36 | 141 | 11 | 22 | 15.60 | 11 | 67 | 279 | 15 | 32 | 11.46 | 19 |
| **Python** | Pyevolve | 52 | 117 | 0 | 0 | 0.0 | 0 | 13 | 54 | 3 | 6 | 11.11 | 3 | 25 | 123 | 4 | 11 | 8.94 | 13 |
| | Ocemp | 19 | 50 | 0 | 0 | 0.0 | 0 | 24 | 60 | 0 | 0 | 0.0 | 0 | 61 | 167 | 3 | 6 | 3.59 | 3 |
| | Noora | 43 | 114 | 4 | 18 | 15.78 | 37 | 8 | 40 | 0 | 0 | 0.0 | 0 | 26 | 126 | 3 | 10 | 7.94 | 12 |

CC = Number Clone Classes CF = Number of cloned fragments SCC = Number of SPCP Clone classes CR = Count of Rules
SCF = Number of cloned fragments that followed SPCP (i.e., the SPCP clones) PSCF = Percentage of SPCP clones

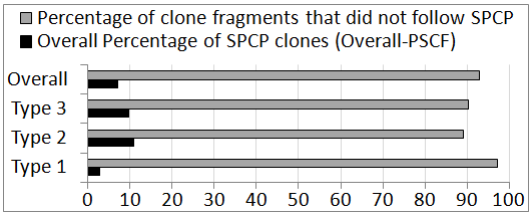


Fig. 5. Overall percentages of cloned fragments that followed SPCP

of SPCP clones of a particular type ($Ti$) of a particular subject system, and $CF_{Ti}$ stands for the total number of clones of type $Ti$ in a particular system. We also calculate the overall percentage of SPCP clones considering all subject systems and all clone-types in a similar way. These overall percentages (Overall-PSCFs) are shown Fig. 5. From this graph it is clear that *a considerable amount (i.e., 97.05%, 89.13%, 90.2%, and 92.96% for Type 1, Type 2, Type 3, and overall case respectively) of clone fragments (i.e., the clone fragments that did not follow SPCP) can be filtered out from our consideration during refactoring because, these cloned fragments either evolved independently or rarely changed. This reduction in the number of considerable refactoring candidates can minimize refactoring effort and cost.*

We answered the following four research questions by analyzing the SPCP clones and rules retrieved by MARC.

*A.* ***Answer to research question RQ 1:*** *Are the changes occurring to the clone fragments in SPCOs (similarity preserving co-changes) consistency ensuring changes?*

To answer this question we manually analyzed 485 SPCOs occurred in the SPCPs of 150 rules (considering Type 3 case of Carol) involving 73 SPCP clones. For each of 100 rules, the support value (SPCO count) was greater than 1 (highest support = 7). Each of the remaining rules had a support of 1.

We know that the clone blocks in each of the rules retrieved by MARC followed an SPCP. The support value of a rule is equal to the number of SPCOs in the SPCP of the rule. While examining the SPCP of a rule our tool stores:

**(1)** The list of commit(s) where the SPCO(s) occurred.

**(2)** Start and end line numbers of each of the participating clone blocks before and after every commit in the above list.

Suppose the clone blocks regarding a particular rule are $CB1$ and $CB2$ respectively and they received an SPCO in commit $C_i$ applied on revision $R_i$. Then, for each of these clone blocks we collect the snapshot in revision $R_i$ and the snapshot in revision $R_{i+1}$ using the line numbers. For each clone block we determine the differences of the corresponding snapshots. Then, we manually compare the changes that occurred to $CB1$ with those that occurred to $CB2$ and decide whether the changes were consistent or not.

**Investigation details.** Among 485 SPCOs that we analyzed manually, in 477 SPCOs (98.35%), the changes to the participating clone blocks (i.e., SPCP clones) were consistent. According to our observation, in each of these 477 SPCOs, the corresponding lines (the same or similar lines) of the two participating SPCP clones were changed in the same or similar way. Thus, the changes in these 477 SPCOs can be termed as consistency ensuring changes to clones. In each of the remaining eight SPCOs, the changes to the clone blocks were not consistent. However, the associated clone blocks still preserved their similarity even after these eight SPCOs.

An example of the consistency ensuring changes that occurred to two clone blocks (corresponding to a rule) in an SPCO on commit 51 of Carol is presented in Fig. 6. The method *bind* and the method *destroySubcontext* in Fig. 6 are Type 3 clones (full method clone) of each other according to the clone detection results of NiCad. From Fig. 4 we see that the rule consisting of these two clone blocks has a support of 7. That is, these two clone blocks received seven SPCOs (i.e., similarity preserving co-changes) in their SPCP (i.e., similarity preserving change pattern). These SPCOs occurred in commits: 42, 51, 54, 58, 91, 153, and 156 respectively. The details of the co-changes in commit 51 (Fig. 6) demonstrate that the changes occurred to the two clone blocks (i.e., method clones), *bind* and *destroySubcontext*, are consistent. As demonstrated in the figure, almost the same if-blocks were added just after the first line in each of these method clones in revision 52. The only differences in these two if-blocks are in the method names and parameters. Also, lines 9 to 12 (in revision 51) in each of the method clones were changed in


Fig. 6. Changes to Two Type 3 Clone Fragments of Carol in commit 51

the same way as can be seen in revision 52. We observed the changes occurred to the method clones in the other six SPCOs too. The changes in each of these SPCOs were also consistent.

We were also interested in identifying the types of changes that mostly occur to the SPCP clones during similarity preserving co-changes (SPCOs). The dominant change types were: addition or deletion of the same or similar statements in both clone fragments; modification of the same or similar corresponding statements in both clone fragments in the same way (e.g., the SPCO occurred in commit 91 on method clones *bind* and *destroySubcontext*); and, addition of the same or similar if-else blocks.

***Answer.*** Thus, *The changes occurring to the SPCP clone fragments in similarity preserving co-changes (SPCOs) generally ensure consistency between the clone fragments. Thus, higher number of SPCOs in a rule indicates higher probability that the participating SPCP clones are related and will also change consistently in future commits. So, ranking of rules for refactoring according to the SPCO count is reasonable.*

### *B. **Answer to research question RQ 2:** Can SPCP clones be candidates for refactoring?*

For answering this research question we manually examined the rules and groups of SPCP clones to determine whether we can suggest particular refactoring for the SPCP clones included in a rule or a group (defined in Section IV-E). For each clone-type of a subject system we considered the top 10 rules and groups (for the cases with less than 10 rules or groups we considered all) totaling 224 rules and 191 groups from all 13 subject systems. Through our manual analysis on these rules and groups, we determine how many of these are refactorable using standard refactoring techniques[1] such as *pull up method*, *extract method*, *remove method*, *parameterize method*, *replace conditional with polymorphism* etc. Then, for each type of clone we determine overall percentage of refactorable rules and groups considering all subject systems. Overall percentages were calculated following a equation similar to Eq.2. Finally, we determine the overall percentage (c.f., Fig. 7) considering all clone-types and subject systems.

Fig. 7 shows that the proportions regarding Type 1 case are the highest ones compared to the other two cases (Type 2, Type 3). As Type 1 clone fragments are exactly similar clone fragments, it was easier to suggest refactoring techniques for them compared to the other two types. Type 2 clones are syntactically similar with variable renaming and/or changes in data types. According to our investigation some Type 2 clones with different data types were not refactorable. We get the lowest proportions for the Type 3 case, because Type 3 clones often had dissimilar code fragment and we could not suggest refactoring for those. Overall, for 64.37% of the association rules (64.62% of the groups), we could suggest refactoring techniques for the participating SPCP clones. Here, we should mention that the groups are formed from the rules (c.f., Section IV-E). As a group may contain more than two SPCP clones while a rule contains only two, the top 10 groups regarding a particular clone type of a particular subject system sometimes had more SPCP clones compared to the corresponding top 10 rules. Thus, the percentages regarding the rules and groups are almost the same with little differences.

However, from our investigation we realize that although we could not suggest refactoring for some rules and groups, the participating clone blocks in these rules and groups might often need to be consistently changed. As we generate XML files containing the rules and groups, these files can suggest us co-change candidates for any future change in any of the SPCP clones. While changing a particular SPCP clone, the developer can look at the other SPCP clones in the same group to determine whether the changes need to be propagated to these clones too. However, we have not yet automated this feature. We also do not investigate the predictability of future co-change candidates in this research work.

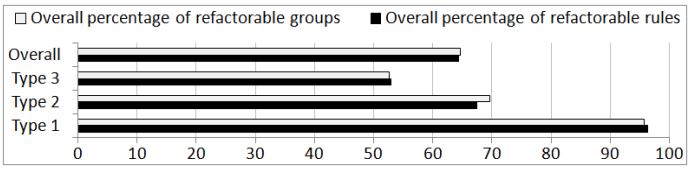


Fig. 7. Overall percentages of refactorable rules and groups

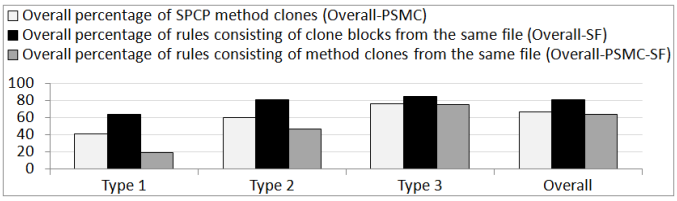


Fig. 8. Statistics regarding SPCP clones and rules

**Example of a refactorable rule.** As an example we can consider the rule (with support / SPCO count = 7) consisting of the method clones, *bind* and *destroySubcontext*, mentioned in Section V-A. The confidence of each of the rules, $bind$ => $destroySubcontext$ and $destroySubcontext$ => $bind$, is one. Thus, these methods (i.e., method clones) always co-changed (i.e., in 7 commits), that means, there is no commit where one changed but the other did not. These methods are almost the same. The only differences are in the method names and parameters (*bind* takes an extra parameter 'obj' of Object type). The first commit where these methods co-changed was applied on revision 42. In this commit, the same statement '*e.prntStackTrace();*' was added after the eighth line of each method. In commit 51 (as mentioned in Section V-A), almost the same if-blocks were added after the first line of each method. Also, the lines 9 to 12 in each of these methods where changed in the similar way (Fig. 6) in this commit. The only differences in the added if-blocks and changed lines were in the names of the methods and parameters. In the commit on revision 54, the same changes occurred at the third line of each method. In the same way, in each of the other commits (58, 91, 153, 156) these two methods received the same changes at the same relative line numbers. The changes obviously indicate that those were made focusing on the consistency of these method clones. Now we discuss the possibility of refactoring these method clones (i.e., merging these into one).

We already mentioned that the method *bind* takes one extra parameter *obj* of type *Object*. The other parameter is the same (same name and type) as that of the method *destroySubcontext*. These two method clones remain in the same file[2] and in the same class (Class Name: *MultiOrbInitialContext*). According to our analysis, it is possible to replace these two methods with a single one that takes the two parameters. The callers of *destroySubcontext* can call it using an extra dummy object. Focusing on this possibility we identified the places where *destroySubcontext* was called. We found two places in the same file[2] where both *bind* and *destroySubcontext* remain (and no other places in the code-base) and determined that we can add an extra dummy object in these calls. This is also possible that the new method that will replace the old ones will take an additional context-sensitive string parameter (a third parameter) for printing purpose. We saw that each of these methods, *bind* and *destroySubcontext*, prints an error message containing the method name. The only difference is the method name used in the message. We can replace this method name by the context sensitive string parameter (that can determine what type of failure has occurred depending on the caller) if necessary. So, the rule consisting of the method clones, *bind* and *destroySubcontext*, is an important refactoring candidate.

**Example of a refactorable group.** An example group of Type 3 clones in Carol contains 4 method clones: *list*, *listBindings*, *rebind*, and *unbind* remaining in the same file[3], under the same package and also in the same class (Class Name: javaURLContext). These method clones are important candidates for refactoring and can be replaced with a single method. Three of these methods (*list*, *listBindings*, *unbind*) perform exactly the same task taking the same parameter. The remaining one takes an extra parameter. It is possible that we decide to only refactor the first three. To refactor all four, one can follow the technique described in our previous example.

***Answer.*** Finally, in answer to the second research question we can say that *a considerable amount of association rules (overall 64.37%) and groups (overall 64.62%) formed by the SPCP clones can be refactored using standard refactoring techniques. Thus, SPCP clones can be considered important candidates for refactoring.* From our experience we realize that the refactoring task often requires proper analysis by the expert users. Also, refactoring can be time consuming because the user might need to analyze the evolution history of the target clones. Thus, automatic identification and ranking of important refactoring candidates (i.e., SPCP clones) can help us minimize refactoring time and effort.

### C. *Answer to research question RQ 3: What are the characteristics of the clones that change following similarity preserving change pattern (SPCP)?*

During manual examination of the rules from Carol we observed that most of the rules consist of method clones (i.e., clone fragments that are full methods). Also, a recent study conducted by Göde [7] demonstrates that developers generally consider removing clones that belong to the same source code file. Considering these two perspectives we determined the followings to answer this research question.

**(1)** The overall proportion of SPCP clones that are full methods (denoted by *Overall-PSMC*).

**(2)** The overall proportion of rules where each rule consists of clones belonging to the same file (denoted by *Overall-SF*).

Overall proportions were calculated using a mechanism similar to the one demonstrated in Eq. 2. After calculating the overall percentages for each clone-type individually, we also determine the overall proportions considering all clone types. Finally, we calculate the overall proportions of rules consisting of SPCP method clones from the same file (denoted by *Overall-PSMC-SF*). These proportions are shown in Fig. 8.

[2]File Path: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java

[3]File Path: carol/src/org/objectweb/carol/jndi/enc/java/javaURLContext.java

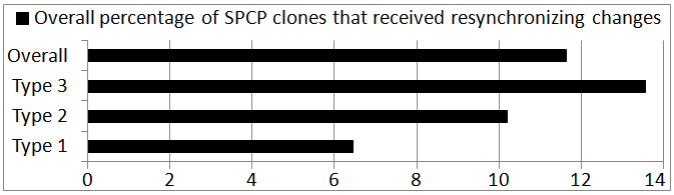

Fig. 9. Proportions of SPCP clones that received resynchronizing changes

From Fig. 8 we see that each of the three measures, Overall-PSMC, Overall-SF, and Overall-PSMC-SF appear in an increasing order from Type 1 case to Type 3 case. Also, Overall-SF is above 60% for each type of clone. Although, Overall-PSMC is below 50% for Type 1 case, this value is above 50% for the other two types with Type 3 case having the highest value (76.66%). From Fig. 5 we see that the percentage of clone fragments that follow SPCP is lowest for Type 1 case. Ultimately, in Fig. 8 the three bars belonging to overall case (considering all clone types and all subject systems) are mainly influenced by Type 2 and Type 3 rules and the values of all three measures are above 60% for the overall case.

***Answer.*** According to our experimental result considering all systems and all three clone-types we can draw the following general conclusions: **(1)** *most of the SPCP clones (overall 66.52%) are method clones* and **(2)** *most of the rules (overall 63.44%) consist of SPCP method clones from the same file.*

In this experiment we detected block clones using NiCad. However, NiCad also facilitates the detection of method clones only. In general, clones in the same file might be easier to be refactored. Clones that belong to different files or folders might require the creation of a separate library for refactoring. This may be difficult without programming language support. Thus, SPCP clones that belong to the same file can be promising refactoring candidates. According to our observation, *we can mainly focus on detecting and refactoring SPCP method clones belonging to the same file.*

### D. *Answer to research question RQ 4: What ratio of the SPCP clones do receive resynchronizing changes?*

We have already mentioned that SPCP clones can receive two types of changes: *similarity preserving changes* and *resynchronizing changes* (elaborated in Section IV-A). Intuitively, resynchronizing changes indicate delayed synchronization among the clone fragments. Delay in synchronization might introduce temporary inconsistency to the functionality of the software systems. We calculated the overall proportions of the SPCP clones that received resynchronizing changes (i.e., delayed synchronizations). For the purpose of calculation we automatically examine the entire evolution histories of the two participating SPCP clones of each association rule and determine whether they received resynchronizing change(s). These proportions are shown in the graph of Fig. 9. We see that overall 6.45%, 10.20%, and 13.57% of the Type 1,Type 2, and Type 3 SPCP clones received resynchronizing changes considering all subject systems. If we consider all subject systems and clone-types, this percentage becomes 11.64%.

***Answer.*** *Thus, according to our investigation, a considerable amount of the SPCP clones can receive resynchronizing changes during evolution. As delay in synchronizations may introduce temporary inconsistencies to a software system, it is important to identify SPCP clones and refactor them.*

## VI. Threats to Validity

In our experiment we detected clones using NiCad [5] clone detector. However, parameter settings of the clone detector can be a threat to our empirical study. For different settings of NiCad, the clone detection results may be different. Thus, there might be variations in the association rules as well as the SPCP clones detected for different settings. Wang et al. [28] defined this problem as the *confounding configuration choice problem* and conducted an extensive study considering six clone detectors to ameliorate the effects of the problem.

However, the settings used in our experiment (as mentioned in our earlier work [17]) are considered standard [19] and with these settings NiCad can detect clones with high precision and recall [20], [21]. Thus, we think that our findings are significant and can help us minimize clone refactoring effort considerably. Moreover, the subject systems that we have used in our experiment are of diverse variety in terms of application domains, implementation languages, sizes and revisions. Thus, we expect that our findings are not biased.

## VII. Related Work

Numerous studies have been conducted regarding the detection, impact analysis [8], [6], [25], [12], [4], [13], [15], [24], [27], management [26], [24] and refactoring [2], [11], [22], [29], [7], [23], [25] of clones. As our experiment is centered on clone refactoring, we discuss the refactoring related studies.

A number of refactoring approaches [2], [11], [22] select clones for refactoring on the basis of the abstract syntax tree representation of the code base. Higo et al. [9] selected clones for refactoring (implementing a tool called CCShaper) based on the lexical analysis of the source code. Zibran and Roy [29] proposed a conflict aware optimal scheduling algorithm for clone refactoring on the basis of constraint programming. They showed that their scheduling algorithm is superior to the other algorithms those are based on genetic algorithm approaches, greedy approaches and linear programming. Bouktif et al. [3] considered the clone refactoring problem as a constrained knapsack problem where the knapsack consists of all the clones to be refactored. They found an optimal schedule for refactoring the clones in the knapsack by applying a genetic algorithm. Göde [7] performed a case study to determine why clones are removed from the code base. According to his observation, developers often consider removing clones residing in the same source code file. Tairas and Gray [23] developed an Eclipse plug-in, CeDAR, that can forward the detected clones to the Eclipse refactoring engine. The Eclipse engine then handles the refactoring decisions. Tsantalis and Chatzigeorgiou [25] proposed a historical volatility model for prioritizing refactoring decisions on the basis of how lately the entities were changed. However, this model is not specialized for clone refactoring.

We see that none of the existing studies and techniques focused on our proposed refactoring step (c.f., Fig. 2) involving the determination of clones that should be considered as important refactoring candidates. Automatic identification of important refactoring candidates can help us minimize refactoring time and effort. We define a particular clone change pattern, SPCP (Similarity Preserving Change Pattern), and show that the clones that changed following this pattern can be important candidates for refactoring. The clones excluded by this pattern either evolved independently or changed rarely during the evolution of the subject system. Thus, these clones should not be our primary targets for refactoring. Our prototype tool MARC can detect all SPCP clones in a system.

## VIII. Conclusion

In this paper, we present an empirical study on the identification of clones that can be considered as important refactoring candidates. We define a particular clone change pattern, SPCP (Similarity Preserving Change Pattern), such that the clones that changed following this pattern during evolution can be considered as important candidates for refactoring. For the purpose of our study, we implement a prototype tool MARC that mines association rules among clones that follow SPCP.

Using MARC we detected all SPCPs in each of our 13 candidate subject systems considering three clone-types (Type 1, 2, and 3). MARC also detects association rules among the SPCP clones and ranks these SPCP clones for refactoring on the basis of the support values of the rules. More importantly, MARC determines groups of refactoring candidates by merging the association rules where a group can contain two or more clones that together followed a SPCP. According to our experimental results and manual investigation, we have the following concluding remarks and suggestions.

**(1)** SPCP clones are important candidates for refactoring. The clones that do not follow SPCP either change independently or are rarely changed. Thus, while taking refactoring decision we suggest to mainly focus on SPCP clones.

**(2)** On an average only 7.04% of the clones existing in a code-base are SPCP clones. Thus, we can filter-out a significant amount (92.96%) of clones from our refactoring decision. We observe that in case of 63.44% of the association rules (formed by the SPCP clones), the two participating SPCP clones are method clones and moreover, these method clones belong to the same source code file. Thus, automatic identification and ranking of SPCP clones can help us minimize a considerable amount of clone refactoring effort and cost. In presence of ranking, we can decide to primarily refactor the more important SPCP clones.

**(3)** According to our manual investigation on 224 rules and 191 groups considering all 13 subject systems, overall 64.37% of the rules and 64.62% of the groups can be refactored using standard refactoring techniques.

**(4)** Refactoring of SPCP clones can minimize the possibility of delayed synchronizations among clone fragments and thus, can minimize unwanted inconsistencies in software systems.

**MARC** is now available on-line [16]. It generates useful XML files containing ranked rules and groups of SPCP clones. We plan to enhance **MARC** to visualize the evolution of SPCP clones. We also plan to explore the possibility of using association rules to predict future co-change candidates.

## 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] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring", Proc. *WCRE*, 2000, pp. 98 – 107.

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

[4] D. Cai, M. Kim,"An Empirical Study of Long-Lived Code Clones", *FASE*, 2011, pp. 432 - 446.

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

[6] N. Göde, Rainer Koschke, "Frequency and Risks of Changes to Clones.", Proc. *ICSE*, 2011, pp. 311 – 320.

[7] N. Göde, "Clone Removal: Fact or Fiction?", Proc. *IWSC*, 2010, pp. 33 – 40.

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

[9] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring Support Based on Code Clone Analysis", *Lecture Notes in Computer Science*, 2004, 3009: 220 – 233.

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

[11] N. Juillerat, and B. Hirsbrunner, "An Algorithm for Detecting and Removing Clones in Java Code", *SeTra*, 2006.

[12] M. Kim, V. Sazawal, D. Notkin, G. Murphy, "An Empirical Study on Code Clone Genealogies", *FSE*, 2005, pp. 187 – 196.

[13] J. Krinke, "A Study of Consistent and Inconsistent Changes to Code Clones", Proc. *WCRE*, 2007, pp. 170-178.

[14] J. Krinke, "Is Cloned Code More Stable than Non-cloned Code?", Proc. *SCAM*, 2008, pp. 57-66.

[15] A. Lozano, and M. Wermelinger, "Assessing the Effect of Clones on Changeability", Proc. *ICSM*, 2008, pp. 227-236.

[16] MARC: https://homepage.usask.ca/~mam815/tools.php

[17] 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.

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

[19] 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.

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

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

[22] S. Schulze and M. Kuhlemann, "Advanced Analysis for Code Clone Removal" *WSR*, 2009.

[23] R. Tairas, and J. Gray, "Increasing Clone Maintenance Support by Unifying Clone Detection and Refactoring Activities", *Information and Software Technology*, 2012, 54(12):1297 – 1307.

[24] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta, "An Empirical Study on the Maintenance of Source Code Clones", *ESE*, 15(1), 2009, pp. 1-34.

[25] N. Tsantalis, A. Chatzigeorgiou, "Ranking Refactoring Suggestions Based on Historical Volatility", Proc. *CSMR*, 2011, pp. 25–34.

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

[27] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei,"Can I Clone This Piece of Code Here?", Proc. *ASE*, 2012, pp. 170 – 179.

[28] 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.

[29] M. F. Zibran, and C. K. Roy, "Conflict Aware Optimal Scheduling of Prioritized Code Clone Refactoring", *IET Software*, 2013, pp. 167 – 186.