Rule-Directed Code Clone Synchronization

Xiao Cheng*, Hao Zhong*‡, Yuting Chen*, Zhenjiang Hu† and Jianjun Zhao*
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†National Institute of Informatics, Japan
{x.cheng, zhonghao, chenyt, zhao-jj}@sjtu.edu.cn, hu@nii.ac.jp

***Abstract*—Code clones are prevalent in software systems due to many factors in software development. Detecting code clones and managing consistency between them along code evolution can be very useful for reducing clone-related bugs and maintenance costs. Despite some early attempts at detecting code clones and managing the consistency between them, the state-of-the-art tool can only handle simple code clones whose structures are identical or quite similar. However, existing empirical studies show that clones can have quite different structures with their evolution, which can easily go beyond the capability of the state-of-the-art tool. In this paper, we propose CCSync, a novel, rule-directed approach, which paves the structure differences between the code clones and synchronizes them even when code clones become quite different in their structures. The key steps of this approach are, given two code clones, to (1) extract a synchronization rule from the relationship between the clones, and (2) once one code fragment is updated, propagate the modifications to the other following the synchronization rule. We have implemented a tool for CCSync and evaluated its effectiveness on five Java projects. Our results shows that there are many code clones suitable for synchronization, and our tool achieves precisions of up to 92% and recalls of up to 84%. In particular, more than 76% of our generated revisions are identical with manual revisions.**

I. Introduction

Software systems commonly contain a large amount of code clones due to many factors in software development and maintenance such as copy-and-paste, refactoring, design patterns, and limitations in the API libraries or frameworks. Empirical studies [22], [19] show that code clones account for 7-23% of the code, and in an extreme case [8], code clones even account for 59% of the code. Code clones place additional burdens to understanding and maintaining code, and attract much attention from both academia and industry. Researchers have proposed various approaches that detect and visualize code clones (*e.g.*, [19], [39]).

Traditionally, code clones are considered harmful (*e.g.*, [3], [15]). Thus, researchers (e.g., [9]) have proposed approaches that remove code clones by refactoring. However, recent empirical studies (*e.g.*, [16], [37]) show that it is tricky to determine whether code clones are harmful or not, since some code clones are beneficial to understanding code and improving software quality. For example, Rajapakse and Jarzabek [31] show that unifying all the code clones can be inefficient and inconvenient for comprehension. As another example, Aversano *et al.* [1] show that refactoring code clones can increase running time. As a result, programmers have to live with code clones, although in many cases, they have to modify code clones simultaneously to maintain consistent changes across the clones.

Researchers have conducted many empirical studies to analyze code clones and their evolution [2], [20], [21], [25], [26], [28]. All of these studies recognized that most of the clone instances in each clone group co-evolve with one another. On the other hand, it is quite difficult to manage code clones, leaving many defects in software. In particular, Jiang *et al.* [13] show that when fixing a bug in a piece of code, programmers may forget to fix the bugs in its clones. Barbour *et al.* [2] also find that programmers can forget to modify some clones, so they have to modify such code in the latter commits. Therefore, it is beneficial to synchronize code clones in time to avoid the buggy inconsistent changes.

In order to manage code clones and eliminate the disadvantages introduced by code clones, researchers have proposed various approaches for managing code clones, synchronizing code clones and keeping their consistency. Zhang *et al.* [36] propose an approach that notifies the programmers when a code clone is modified. Their work allows the programmers to update code clones manually. However, as code clones abound in software projects and it is error-prone to manually update code clones, there is an urgent need for a tool that can update code clones automatically. Nguyen *et al.* [30] propose an approach that synchronizes clones whose Abstract Syntax Trees (ASTs) are identical or with minor differences. When synchronizing clones, their approach records edits on an AST of a code fragment and applies recorded edits on its clones. Kim *et al.* [17] show that most clones become less similar to each other along the evolution of software. When this happens, the state-of-the-art approach [30] is insufficient to keep the consistency of clones with diverging code clones, due to the following challenges:

Challenge 1. The change rules of clones can become complicated. When clones become less similar, their ASTs can become quite different. As a result, it becomes challenging to record AST changes from a piece of code and apply the recorded changes to its clones as Nguyen *et al.* [30] do.

Challenge 2. When being maintained, code fragments can be similar, but may not need to be changed simultaneously. When the state-of-the-art tool [30] records and applies code changes, it becomes quite challenging to distinguish those changes from the ones that need to be changed simultaneously, and the

‡Corresponding author

978-1-5090-1428-6/16/$31.00 ©2016 IEEE

TABLE I
A CLONE-SYNCHRONIZATION RELATED BUG

ci_1

```
1 private int getColumn(Point point){
2   int colCnt = fTableViewer.getTable().
        getColumnCount();
3   TableItem item = null;
4   for (int i=0; i<fTableViewer.getTable().
        getItemCount(); i++){
5     item = fTableViewer.getTable().getItem(i);
6   }
7+  if (item != null){ // a check here
8     for (int i=0; i<colCnt; i++){
9       Point start = new Point(item.getBounds(i).x,
            item.getBounds(i).y);
10      ...
11      if (start.x < point.x && end.x > point.x)
12        return i;
13    }
14+ }
15  return -1;
16 }
```

ci_2

```
1 private int getColumn(Point point){
2   int colCnt = fTableViewer.getTable()
        .getColumnCount();
3   TableItem item = fTableViewer.
        getTable().getItem(0);
4   for (int i=0; i<colCnt; i++){
5     Point start = new Point(item.
          getBounds(i).x, item.getBounds(
          i).y);
6     ...
7     if (start.x < point.x && end.x >
          point.x)
8     return i;
9   }
10  return -1;
11 }
```

ci_2'

```
1 private int getColumn(Point point){
2   int colCnt = fTableViewer.getTable()
        .getColumnCount();
3   TableItem item = fTableViewer.
        getTable().getItem(0);
4   if (item != null){ // a check here
5     for (int i=0; i<colCnt; i++){
6       Point start = new Point(item.
            getBounds(i).x, item.
            getBounds(i).y);
7       ...
8       if (start.x < point.x && end.x >
            point.x)
9       return i;
10    }
11  }
12  return -1;
13 }
```

synchronized code can be incorrect.

If the consistency of the code clones are not carefully maintained, it can introduce bugs in the code under development. For example, we have found a clone-related bug in Eclipse[1]. Table I shows the involved two methods (ci_1 and ci_2), and they are code clones with different ASTs at the method level. In particular, ci_1 is taken from `AbstractAsyncTableRendering.java`, and ci_2 is taken from `AbstractTableRendering.java`. Line 5 of ci_1 (or line 3 of ci_2) obtains an `item` by calling the method `getTable().getItem()`. In a previous buggy version of ci_1, this line leads to an exception, when `item` is `null`. The bug was reported, and in Line 7 of ci_1, programmers fix the bug by calling an extra check. However, the bug is not fully fixed, since its code clones are not synchronized accordingly. For example, Line 5 of ci_2 can still throw exceptions, when `item` is `null`. With a proper clone management tool, we believe, the above bug can be fully fixed, since the fixes in ci_1 can also be applied to ci_2. However, the state-of-the-art tool [30] is insufficient to synchronize the above two methods, since their ASTs are different.

In this paper, we propose a novel rule-directed approach called CCSync. Instead of recording changes in ASTs of code clones, CCSync generates synchronization rules and allows programmers to tailor such rules. With tailored rules, CCSync propagates changes on a code construct to its *allelic* code construct that locates at a comparable position in other clones. Compared with Nguyen *et al.* [30], CCSync is more general, since it allows synchronizing clones no matter whether they have identical structures or not. This paper makes the following contributions:

- We formally define synchronization rules for keeping the consistency of code clones. A synchronization rule describes the relation between two allelic code constructs and the strategy to propagate changes from one code construct to its allelic one.
- We propose a novel rule-directed approach that synchronizes code clones. Our approach includes (1) an algorithm for generating synchronization rules from code clones and allowing customization by the programmers; and (2) an algorithm for utilizing such rules to direct the propagation of code changes in appropriate positions.
- We implemented a tool for CCSync, and with its support, we conducted an evaluation on five projects. The results show that our tool achieves precisions of up to 92% and recalls of up to 84%. In particular, more than 76% of our generated revisions are identical with manual revisions.

The rest of the paper is organized as follows: Section II presents the terminologies on code clones. Section III utilizes a running example to illustrate how CCsync works. Section IV presents the details of our approach. Section V evaluates CCSync. Section VI discusses related issues. Section VII presents the related work. Section VIII concludes.

II. TERMINOLOGY

In this section, we introduce the terminologies briefly.

Code Fragment. A *code fragment* (cf) is a sequence of code lines such as code statements.

Code Clone. Two pieces of code fragments (cf_1 and cf_2) are code clones, if there exists a similarity function f and $f(cf_1, cf_2) < \sigma$. For example, Jiang *et al.* [12] define the similarity function as the edit distances between two pieces of code fragments. Here, σ is a predefined threshold. We use $\langle cf_1, cf_2 \rangle$ to denote that cf_1 and cf_2 are code clones, and call it a *clone pair*. If $\langle cf_i, cf_j \rangle$ holds for $cf_1, \ldots, cf_n$, $\langle cf_1, cf_2, \cdots, cf_n \rangle$ is called a *clone group*.

Clone Instance. A code fragment cf_i in a clone pair or a clone group is called a *clone instance* (ci).

Clone Type. We follow the definition of Bellon *et al.* [4], and define the following three types of code clones:

- **Type I** All the clone instances are identical with modifications only on layouts and comments.
- **Type II** All the clone instances are syntactically identical with modifications on only names of identifiers, types, literals, and methods.
- **Type III** All the other code clones, where clone instances can have further modifications (*e.g.*, inserting, deleting, and updating statements).

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=174722

III. Running Example

First, let us revisit the example in Section I. We use this example to further illustrate the problem and how our approach works. Table I shows the two clone instances ci_1 and ci_2 of the clone pair. By the definition in Section II, ci_1 and ci_2 are Type III code clones, since they are not syntactically identical. As the ASTs of the two methods are different, it is challenging to simply record changes in the AST of ci_1 and apply those changes on that of ci_2. As a result, the state-of-the-art tool [30] cannot synchronize the code clone.

Instead of recording and applying changes in ASTs, our key insight is that we can build the mapping relations between common code fragments directly, and update a code fragment when its mapped code fragments are edited. For this example, after we build the mapping relation between Line 8 in ci_1 and Line 4 in ci_2, when programmers add a check in Line 7 of ci_1, we can add the same check before Line 4 in ci_2 to avoid the same buggy behavior. The synchronized code is shown as ci_2' in Table I, where a check is added correctly.

To generate the above synchronized code, CCSync first identifies the mapping relation between two clone instances. In particular, the `for` loop in ci_1 (Line 8 to Line 13) is aligned to the `for` loop in ci_2 (Line 4 to Line 9). The two `for` loops are called *node constructs* and consist of a *construct multiset* (see in Section IV-B). According to the rule that contains the construct multiset and attached action set, the aligned identical constructs (*e.g.*, the `for` loops) should be changed simultaneously (*e.g.*, inserting the same check before them). The detailed algorithm in the approach will be presented in Section IV-C and Section IV-D.

IV. Approach

In this section, we present the overview of our approach (Section IV-A), our definition of synchronization rules (Section IV-B), the steps of generating such rules (Section IV-C), the steps of synchronizing with such rules (Section IV-D), and our extension for clone groups (Section IV-E).

A. Overview

The overview of our CCSync approach is presented in Figure 1. Each rectangle represents an entity in the synchronization steps, and each ellipse represents an action or an operation that connects two entities. The inputs of CCSync are: (1) the project under development or maintenance, which contains code clones, (2) changes in a clone instance, and (3) the choices of tailoring the actions in synchronization rules. The output is a set of synchronized clone pairs (or groups). The whole process is divided into two main parts, *i.e.*, rule generation and rule-directed change propagation:

1) **Rule Generation.** This part aims to identify synchronization rules automatically from known code clones to reduce the effort of writing them. Meanwhile, a user can also customize the rule (Section IV-C).
2) **Rule-Directed Change Propagation.** This part aims to utilize the rules to propagate changes (*i.e.*, updating, deleting, or inserting the corresponding code constructs in appropriate positions) from one clone instance to the other clone instance (Section IV-D).

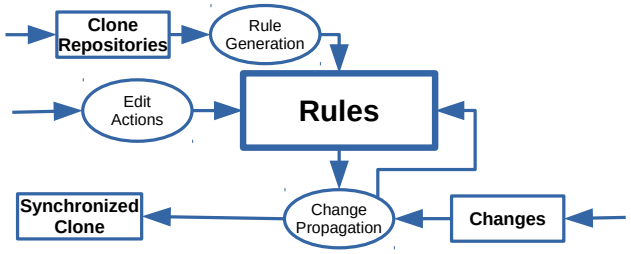

Fig. 1. Overview of CCSync

We understand that existing tools cannot effectively identify the exact intention for changes in Type III code clones, so we allow manual edits of synchronization rules to accommodate code clones with significant differences. In addition, although CCSync mainly focuses on synchronization of clone pairs, we discuss its extensibility for clone groups in Section IV-E.

B. Synchronization Rule

Before we introduce synchronization rules, we define the basic units of the synchronization, *code constructs*, as follow:

Definition 1: ***Code Construct.*** *A* ***code construct*** *is a token in a code fragment (****token construct****) or a sequence of tokens in a code fragment, which can be parsed as a node in the parse tree (****node construct****), or* ϵ*,* i.e.*, a placeholder code construct (*ϵ ***construct****),* e.g.*,*

$$construct \quad ::= \quad token \mid node \mid \epsilon$$

In this paper, *code construct* is shortened as *construct*.

Lin *et al.* [24] have proposed an approach, called *MCIDiff*, to detect differences across clone instances. They use multisets to represent the results. We follow the term multiset and extend it from tokens to code constructs for the ease of synchronization by defining *construct multiset*.

Definition 2: ***Construct Multiset (CM).*** *A* ***construct multiset*** *is a pair (or a group) of corresponding code constructs (one from each clone instance). A code construct from a clone instance can only be in one multiset. If all pairs of corresponding constructs in a CM are identical (*i.e.*, they have exactly the same category and attribute), the CM is a matched CM. Otherwise, it is a differential CM.*

Both the matched CM and differential reflect the corresponding relation among the construct from each clone instance. The corresponding constructs in the CM are called **allelic constructs** of each other.

Figure 2 shows a pair of code construct sequences of the method body in two clone instances ci_1 and ci_2 in Table I. Here, we omit the check statement in ci_1. Each row of rectangles contains a pair of allelic constructs and it represents a *CM*. Different colors of the rectangles illustrate different categories of construct multiset. The light blue ones (*e.g.*, CM_1 or CM_2) denote that the matched CM. For the dark blue ones (*e.g.*, CM_5 or CM_6), the constructs on the left side are real constructs (*i.e.*, a token or a sequence of tokens), and the constructs on the right side are ϵ. Here, the ϵ construct denotes

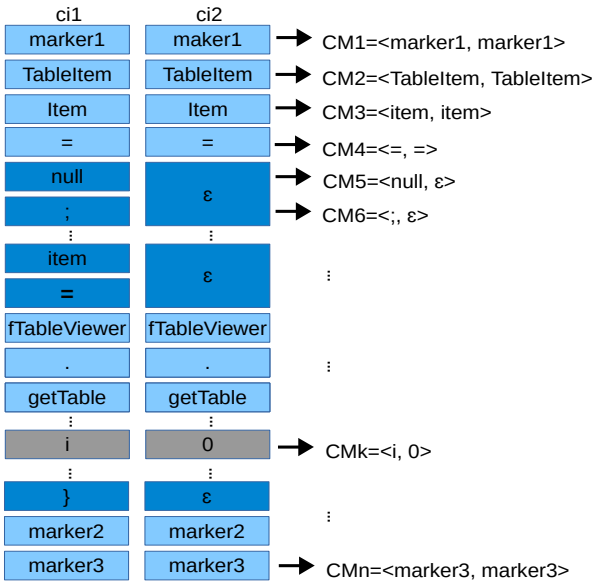

Fig. 2. Example of construct multisets

that no real construct from a clone instance corresponds to the real construct from the other clone instance. The gray ones (*e.g.*, CM_k) denote that the allelic constructs from different clone instances are not identical but similar.

To compress long sequences of tokens, we use the $marker_i$ which is mapped from the node construct as an identification to denote node construct itself. For example, the two $marker_1$s in CM_1 denote the assignment statements in line 2 of ci_1 and ci_2 respectively in Table I:

```
int colCnt = fTableViewer.getTable().getColumnCount();
```

The $marker_2$ and $marker_3$ in Figure 2 denote the `for` loop (*i.e.*, Line 8 to Line 13 in ci_1) and the `return` statement (*i.e.*, Line 15 in ci_1) respectively.

Based on the concept of construct multisets, we define synchronization rules as follows:

Definition 3: ***Synchronization Rule.*** *A* ***synchronization rule*** *is a pair* $\langle CM, action^\star \rangle$*, where* CM *is a construct multiset and* $action^\star$ *is a set of synchronization actions on* CM*:*

$$rule \quad ::= \quad CM, action^\star$$

Synchronization rules have two roles in our approach. First, for the clone instances of each code clone, a synchronization rule defines the mappings between their constructs. CCSync aligns the token constructs in a *CM* by their category and attributes. As each node construct is mapped with a marker, CCSync aligns node constructs, if their markers are identical. CCSync aligns the remaining constructs to ϵ constructs (placeholder) at proper locations. Here, it decides the location of an ϵ construct by comparing its already mapped neighbor constructs. Second, the right hand side of a synchronization rule is a set of actions to be applied to a *CM* when any of its two code constructs changes (including the gaps to the immediate neighboring *CMs* - *i.e.,* insertions before and after *CM* in the ordered list of *CMs*). Table II lists a collection of candidate actions, which can meet the basic requirement of change propagation. If the matched CM contains the exactly identical constructs, CCSync attaches all actions in Table II to it. For example, a rule for the first CM in Figure 2 is represented as "$\langle marker_1, marker_1 \rangle$, {i, ii, iii, iv, v, vi, vii, viii}", where the roman numerals reference the actions in Table II. This indicates that the identical constructs should be changed exactly alike. In particular, If the statement represented by $marker_1$ is changed to

```
int colCnt = fTableViewer.getTable().getColumnCount() - 1;
```

The allelic constructs of ci_1 in ci_2 will be changed accordingly.

CCSync allows customizing synchronization rules, and programmers can enable and disable the actions in a synchronization rule. For example, a programmer can omit some changes on a construct if they disable corresponding actions. Identifying whether a change need to be propagated or not is not discussed in the paper and left to be the future work.

TABLE II
ACTIONS

No.	Action	Description
i	$left \rightarrow right$	Given a *CM*, when the left construct is changed to $left'$, the right one will be replaced by $left'$.
ii	$left \leftarrow right$	Given a *CM*, when the right construct is changed to $right'$, the left one will be replaced by $right'$.
iii	$\overline{left} \rightarrow \overline{right}$	Given a *CM*, when a construct is inserted before the left construct, the same construct will be inserted before the right construct to form a new *CM′* before the given *CM*.
iv	$\overline{left} \leftarrow \overline{right}$	Given a *CM*, when a construct is inserted before the right construct, the same construct will be inserted before the left construct to form a new *CM′* before the given *CM*.
v	$\underline{left} \rightarrow \underline{right}$	Given a *CM*, when a construct is inserted after the left construct, the same construct will be inserted after the right construct to form a new *CM′* after the given *CM*.
vi	$\underline{left} \leftarrow \underline{right}$	Given a *CM*, when a construct is inserted after the right construct, the same construct will be inserted after the left construct to form a new *CM′* after the given *CM*.
vii	$left\star$	Given a *CM*, when the left construct is changed to $left'$, all the constructs in its clone instance which have the same marker will be changed to $left'$.
viii	$right\star$	Given a *CM*, when the right construct is changed to $right'$, all the constructs in its clone instance which have the same marker will be changed to $right'$.

C. Rule Generation

Figure 3 shows the process of rule generation. To generate synchronization rules, CCSync has the following three steps:

1) producing the construct multisets;
2) selecting a set of actions for each multiset;
3) customizing the rules by enabling/disabling the actions for the concerned multisets (optional).

To produce the construct multisets, CCSync first transforms the source code of each clone instance to a sequence of tokens, and generates an initial sequence pair. Due to the similarity between clone instances, we find that subsequences of tokens can appear frequently in each sequence of the sequence pair. When synchronizing clones, programmers often need to maintain the consistency of them. These subsequences

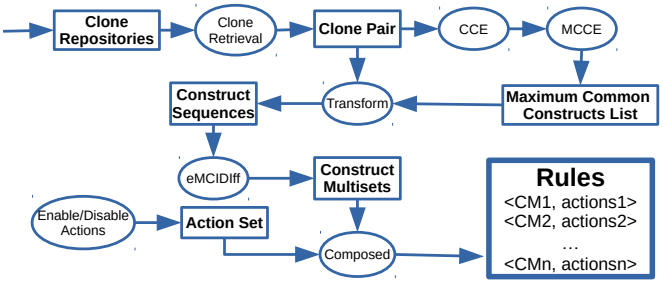

Fig. 3. Process of rule generation

of token constructs may be parsed into nodes or subtrees of an AST. CCSync merges these tokens into a node construct. As these constructs appear in all the sequences, we call them *common code constructs*. For example, the `for` loop in ci_1 (Line 8 to Line 13) appears in ci_2 (Line 4 to Line 9) in Table I, so the `for` loops are common code constructs.

A large common construct may cover some small common constructs. For example, all the expressions (*e.g.*, `i = 0`) and statements (*e.g.*, `return i;`) in the above `for` loops are also common constructs, since the `for` loop is a common construct. If a common construct is not covered by any other common constructs, we call it a *maximum common construct*. It is more convenient to maintain the maximum common constructs, since aligning larger common constructs can reduce the mismatches. For example, in Table I, "`i=0`", "`i < colCn`" and "`i++`" (Line 8 in ci_1 and Line 4 in ci_2) are identified as common code constructs, and the whole `for` loops (line 8 to line 13 in ci_1 line 4 to line 9 in ci_2) are also identified as a maximum common constructs. The latter ones cover the former ones. Aligning two `for` loops can prevent from mismatching "`i = 0`" in Line 4 of ci_1 to the same construct in Line 4 of ci_2. As a result, in the construct multiset, CCSync detects the common construct as larger as possible.

Given a clone pair, CCSync first identifies their common code constructs, and then excludes the small ones that covered by larger ones. Algorithm 1 describes how to extract common constructs from a clone pair. The input of Algorithm 1 is a pair of ASTs such as *left* and *right*.

The algorithm has the following two steps:

1) It firstly classifies all of the nodes of each AST according to their hash code of content (Line 1 to Line 11). The nodes in the same list have the same hash value, which implies they contains the same source code.
2) For each node list, it checks whether the nodes come from *left* or *right* are identical or not. If these nodes are found only in one AST, they will be excluded (Line 12 to Line 16). The remaining lists contain all of the common constructs (Line 18).

Algorithm 2 refines the found common constructs ccs to obtain maximum common constructs $mccs$. The algorithm visits all of the nodes in *left* and *right* starting from their roots to selected maximum common code constructs.

CCSync builds mappings from maximum common constructs to markers according to the source code covered

Algorithm 1: Common Constructs Extraction (CCE)

Input: $AST\ left, right$
Output: $List\ ccs$ // a list of code constructs

```
1  List cList ← all nodes in left and right;
2  Map hc2cList ← φ; // a map from hash code to construct list
3  foreach node ∈ cList do
4      if !hc2cList.containKey(node.hashCode) then
5          List list ← φ;
6          list.add(node);
7          hc2cList.put(node.hashCode, list);
8      else
9          hc2cList.get(node.hashCode).add(node);
10     end
11 end
12 foreach (hashCode, list) ∈ hc2cList do
13     if list ∩ left = φ || list ∩ right = φ then
14         hc2cList.remove(hashCode);
15     end
16 end
17 List ccs ← hac2cList.valSet;
18 return ccs;
```

Algorithm 2: Maximum Common Constructs Extraction (MCCE)

Input: $List\ ccs, AST\ left, right$
Output: $List\ mccs$

```
1  Queue q ← φ;
2  Lis mccs ← ccs;
3  q.add(left.root);
4  q.add(right.root);
5  while q ≠ φ do
6      ASTNode node ← q.poll();
7      if node ∈ ccs then
8          mccs.add(node);
9      else
10         q.add(node.children);
11     end
12 end
13 return mccs;
```

by common constructs. In particular, CCSync replaces each maximum common construct with a marker (*e.g.*, $\$\_number$). The remaining parts, which exclude the common constructs, are tokenized as token constructs.

CCSync includes an algorithm, called *eMCIDiff*, that extends the *MCIDiff* algorithm [24] to generate construct multisets. *MCIdiff* generates a sequence of tokens for each clone instance, and then computes the Longest Common Subsequence (LCS) of generated token sequences. Based on the LCS, *MCIDiff* identifies matched and unmatched tokens. It is inconvenient to maintain the consistency among tokens, since tokens provide limited information on syntaxes. Instead of the token level, CCSync extends the *MCIDiff* to analyze clone instances at the construct level. Furthermore, CCSync introduces an additional node construct category, called *Marker*, to define mapping relations between common code constructs. CCSync assigns each code construct to a marker. During the process, it aligns code constructs by their categories such as *Type*, *Method/Filed/Variable/Literal*, *Label*, *Keyword*, *Separator*, and *Operator*. As each code construct is mapped with a marker, code constructs are matched if their markers are identical.

Algorithm 3: Multiset Generation

Input: *List seqs* // a list of construct lists
Output: *List multisets*

```
1 List constructs ← ∅;
2 foreach seq ∈ seqs do
3 |   constructs.add(seq);
4 |   ranges.add([0 ∼ seq.size() − 1]);
5 end
6 List multisets ← eMCIDiff(constructs, ranges);
7 return multisets;
```

Markers thus can prevent the subsequences of code constructs from being mismatched. Algorithm 3 calls *eMCIDiff* to generate construct multisets. Due to space limitation, we omit its details, since its general concept is similar to *MCIDiff* [24].

CCSync adopts a conservative strategy to infer rules and attaches some actions for a rule by default. For example, it attaches Actions i to viii in Table II to the construct multisets which consist of node constructs, and Actions iii to vi to the construct multisets which consist of token constructs or ϵ constructs. CCSync allows customizing rules since code fragments in clones may not change simultaneously. If programmers determine that a modification of a code fragment shall not be propagated to its clone instances, they can disable the actions on the code fragment during synchronization. The programmers can later enable the actions. For example, if programmers decide that the `for` loops in ci_1 and ci_2 in Table I shall be evolved independently, they can disable the actions attached to the `for` loop, and then the changes will not be propagated.

D. Rule-Directed Change Propagation

Figure 4 shows the process of propagating changes from one clone instance to the other clone instances, under the guide of synchronization rules. CCSync utilizes *eMCIDiff* to compare the modified clone instance with the original one, and identifies modification actions such as updating, deleting, and inserting. After that, CCSync looks up the corresponding rules for modified constructs and propagates modification actions to their allelic constructs accordingly. After synchronization, CCSync refreshes rules, if modification actions change their related multisets. Particularly, if there are two subsequent CMs: CM_1 and CM_2, where CM_1 has an insertion rule for additions after CM_1 and CM_2 has an insertion rule for additions before CM_2. If X is added in between CM_1 and CM_2. CCSync will check the actual actions of consecutive CMs to avoid inserting X twice.

For example, in Table I, CCSync detects that the check in Line 7 and the right curly bracket in Line 14 of ci_1 are inserted constructs before and after the `for` loop, respectively. The rule for the `for` loop contains Action iii and Action v that directly insert the same check before and the right curly bracket after the allelic `for` loop in ci_2. In this way, the bug is fully fixed. Meanwhile, the rule before the `for` loop contains Aaction vthat also directly insert the same check before the `for` loop.

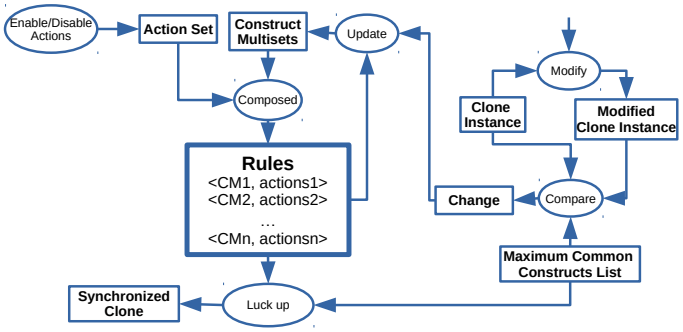

Fig. 4. Process of rule-directed change propagation

However, CCsync finds that the two actions of the consecutive rules has the same effect, it only inserts the check once.

To generate the synchronized code, CCSync prints the constructs sequences in order and formats them appropriately: for the token construct, CCSync prints it directly; for the node construct (marker), CCSync looks up the *Maximum Common Construct List* to get the source code of these node constructs and prints them; CCsync skips all ϵ constructs.

E. Extension for Clone Groups

We further extend CCSync to support to synchronize clone groups, whose clone instances are more than two. The key point to support synchronization for groups is the generation of synchronization rules:

1) generating construct multisets consisting of n code constructs (one from each clone instance).
2) assigning actions to each construct multiset.

We extend Algorithm 1 and 2 for more than two clone instances and our extended CCSync searches for common constructs and maximum common constructs from more than two clone instances. For clone pairs, common constructs appear in both clone instances, while for clone groups, common constructs appear in all clone instances. As MCIDiff [24] supports representing differences in more than two clone instances, we easily extend Algorithm 3 to get constructs multisets consisting of code constructs from more than two clone instances.

In addition, as each clone instance has a construct multiset that consists of n code constructs, for any pair of $\langle c_i, c_j \rangle$ $(1 \leq i, j \leq n)$ in a construct multiset $\langle c_1, c_2, \cdots, c_n \rangle$, our extended CCSync attaches a set of actions in Table II to define the strategy when a change occurs in one of the clone instances.

V. Evaluation

We have implemented CCSync, and conducted an evaluation to answer the following three research questions:

- RQ1: How many opportunities are there to synchronize code clones?
- RQ2: How effectively does CCSync synchronize clone pairs with regard to real-world changes?
- RQ3: How effectively does CCSync synchronize clone groups, with our extension?

In Section V-A, our results show that in real code, there is a strong need for synchronizing clones. In Section V-B, our

TABLE III
CHARACTERISTICS OF SUBJECT PROJECTS

Projects	#Revision	#File	#LoC	#Pairs	#LoCC	#Co-change
jEdit	2,501 - 3,000	366	76,993	341	16,251	377
JFreeChart	1 - 500	947	128,118	819	39,923	312
JHotDraw	1 - 500	1,545	126,767	595	9,556	1,128
Columba	1 - 464	2,253	147,254	641	23,981	349
OSWorkflow	101 - 600	288	24,140	163	8,532	197
Total	2,964	5,399	503,272	2,559	98,243	2,363

results show that CCSync achieves both high precisions and recalls when utilizing the rules to synchronize clones in real code In Section V-C, our results show that CCSync is able to conditionally support the synchronization of clone groups, especially when manually tailoring the generated rules.

A. RQ1: The Need for Synchronizing Clones

1) Clone Detection Tool and Subjects: In this study, we utilized ConQAT [14] to detect clones. In particular, we set the *gapped ratio* of ConQAT as 0.5, the *minlength* of clones as 10, and the *max* of errors as 5. The arguments indicates that if the detected clone pair has twenty statements, at least its five statements in a clone instance are different from those in the other clone instance. With the arguments, most detected clone pairs are Type III clones. Wang *et al.* [34] proposed an approach to searching the configuration space of clone detection tools appropriately, however, we consider the the extensive approach is not necessary in our experiment. We chose five large open source projects, *i.e.*, jEdit, JFreeChart, JHotDraw, Columba and OSWorkflow, as our subjects. From the project repository of each project, we selected 500 revisions. Table III shows the projects, and it lists the range of analyzed revisions, the number of files, the lines of code (LoC), the number of detected clone pairs, and the LoC of detected clones, column by column. For JEdit, we could not access its revisions before 2500, so we collected its revisions since 2500. For Columba, we collected all its revisions, and it had only 464 revisions when we conducted the evaluation.

2) Detecting Co-change Clone Pairs: For each detected clone pair in a commit $\langle ci_1, ci_2 \rangle$ and in the next commit $\langle ci'_1, ci'_2 \rangle$, we considered it as a co-change clone pair, if ci_1 and ci_2 were both changed. We counted the number of times that co-changes occurred along the projects revision history. We considered these co-changes as cases where clone synchronization is needed as synchronizing them can reduce manual edits needed for the clones.

3) Result: Figure 5 shows the results of detected clone pairs. Its horizontal axis denotes revisions, and its vertical axis denotes detected clone pairs. The results show that with the evolution of the five projects, more clone pairs are detected. Figure 6 shows the results of the accumulated number of times that co-changes occur. Its horizontal axis denotes revisions, and its vertical axis denotes the accumulated number of times that co-changes occur for all detected clone pairs, which reflect how often co-changes occur in the detected clone pairs.

To measure how often synchronization opportunities occur for a clone pair on average along software evolution, we divide the accumulated number of times that co-changes occur by the number of the detected clone pairs and build Figure 7. The results indicates the usefulness of CCSync with the evolution process. For example, the last point of the jHotDraw curve shows that on average, each clone pair in jHotDraw has 1.8 times of co-change. The last column of Table III lists total number of co-change times, which reflects the opportunities of an approach like CCSync.

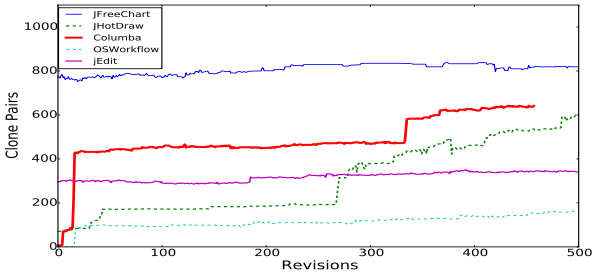

Fig. 5. Number of detected clone pairs

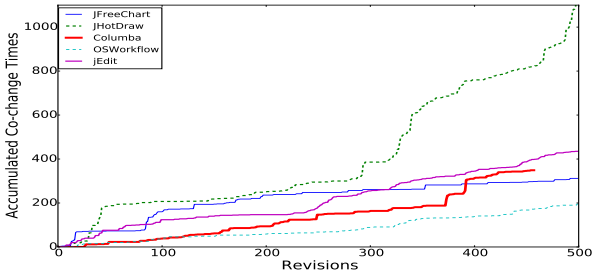

Fig. 6. Accumulated number of times that co-changes occur

We randomly sampled 100 co-changes. We found that more than half of such co-changes occur in the clone pairs whose ASTs are different, and cannot be effectively handled by recording and applying such changes in ASTs.

B. RQ2: Synchronizing Real Changes

For this research question, we evaluated the effectiveness of our approach with real changes of detected clone pairs.

1) Setup: In a repository, if programmers modified a clone pair $\langle ci_{1i}, ci_{2i} \rangle$ in Revision i to a clone pair $\langle ci_{1j}, ci_{2j} \rangle$ in Revision j ($i < j$), we extracted the changes from ci_{2i} to ci_{2j}, and called them as **Manual Changes** (MC). We fed the changes from ci_{1i} to ci_{1j} to CCSync. It automatically modified ci_{2is} to ci_{2js}, and we called the changes as **Sync Changes** (SC). For each modified clone pair, we calculated precisions and recalls as follows:

$$precision = \frac{\#(MC \cap SC)}{\#SC} \qquad recall = \frac{\#(MC \cap SC)}{\#MC}$$

We used precisions, recalls, and their distribution to measure the effectiveness of CCSync. We counted MCs and SCs for all the clone instances for each project, and then counted clone instances whose precisions and recalls fell into specific ranges to calculate their distribution.

In this evaluation, we employ *ChangeDistiller* [10] to compare original revisions with modified revisions for MCs and SCs. If compared revisions contain syntax errors, *ChangeDistiller* often throws exceptions and produces no changes. As a

TABLE IV
RESULTS OF SYNCHRONIZING REAL CHANGES

Project	#CI	#MC	#SC	#∩	P	R	I	P1	P2	P3	R1	R2	R3
jEdit	598	3,876	3,283	2,994	92.0%	77.2%	71.1%	21.4%	6%	72.6%	17.9%	2.8%	79.3%
JFreeChart	340	2,080	1,852	1,765	95.3%	84.9%	75.3%	13.5%	7.1%	79.4%	12.9%	4.7%	82.4%
JHotDraw	1,830	13,257	12,331	11,418	92.6%	86.1%	76.6%	15.6%	5.8%	78.6%	10.8%	4.4%	84.8%
Columba	450	1,854	1,819	1,682	92.5%	90.7%	81.6%	10.7%	5.6%	83.7%	8.7%	3.8%	87.5%
OSWorkflow	266	1,526	1,513	1,277	84.4%	83.6%	74.4%	13.5%	9.4%	77.1%	10.2%	5.3%	84.5%
Total	3,484	22,593	20,798	19,136	92.0%	84.7%	76.0%	15.6%	6.2%	78.2%	11.9%	4.1%	84.0%

#CI: #clone instances; #∩: #(MC ∩ SC); P: Precision; R: Recall; I: Identical; P1: $[0, 0.5)$; P2: $[0.5, 1)$; P3: 1; R1: $[0, 0.5)$; R2: $[0.5, 1)$; R3: 1.

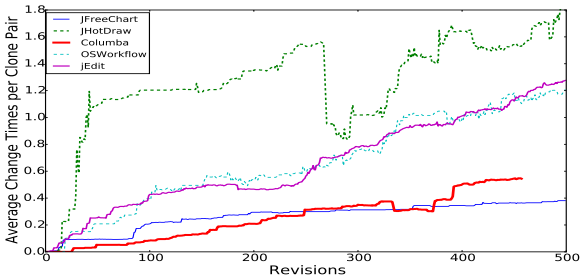

Fig. 7. Average number of times co-changes occur per clone pair

result, if we identify a MC and a SC are identical, the two changes are identical, and do not introduce compilation errors.

2) Result: Table IV shows the results.

Column "#CI" lists the number of clone instances in all revisions for each project. Due to the change of file name, class name, or method name, some clone pairs cannot be matched in the next revision and the MC cannot be extracted, so we exclude these clone instances in each revision. That's why the number of clone instances here is not coincident with the number of co-change clone pairs in Table III. Column "#MC" lists the number of extracted Manual Changes of all revisions. Column "#SC" lists the number of Sync Changes of all revisions. Column "#∩" lists the number of identical changes between MCs and SCs of all revisions.

Columns "P" and "R" show precisions and recalls, respectively. In total, the precision and the recall are 92% and 84.7%, respectively. That is to say, most SCs are quite similar with MCs. However, a clone instance can have multiple changes to be synchronized, and it is essential to understand how many synchronized clone instances are identical with manually changed code. Column "I" shows ratios of clone pairs whose SCs are identical to MCs. In total, the results show that in 76% of the clone pairs, SCs are identical with MCs.

Columns "P1", "P2" and "P3" show the distribution for precisions. In particular, Column "P1" lists the ratios of the clone instances whose precisions are below 50%. Column "P2" lists the ratios of the clone instances whose precisions are between 50% and 99%. Column "P3" lists the ratios of the clone instances whose precisions are 100%. We find that CCSync achieved high precisions (100%) on 78.2% of clone pairs; middle precisions (50% to 99%) on 6.2% of clone pairs; and low precisions (below 50%) on 15.6% of clone pairs.

Columns "R1", "R2" and "R3" show the distribution for recalls. Column "R1" lists the ratios of the clone instances whose recalls are below 50%. Column "R2" lists the ratios of the clone instances whose recalls are between 50% and 99%. Column "R3" lists the ratios of the clone instances whose recalls are 100%. We find that CCSync achieved high recalls (100%) on 84.0% of clone pairs; middle recalls (50% to 99%) on 4.1% of clone pairs; and low recalls (below 50%) on 11.9% of clone instances.

In summary, our results show in total, 76% of synchronized clone instances whose SCs are identical with MCs. In particular, CCSync achieved 100% precisions or recalls on many clones instances, *i.e.*, 72.8% or 84% respectively. The results highlight the effectiveness of CCSync on synchronizing real-world clone pairs.

3) Discussion: Still, our tool achieves low precisions and recalls (<50%) on a small portion of clone pairs (about 10%). After manual inspection, we identify the following issues:

1. Their synchronization rules conflict with programmers' intention. For example, when programmers insert a statement to a clone instance, they may not insert the same statement to the other clone instances.
2. The clone instances of a clone pair are not co-changed at the same revision but are co-changes in the latter revisions. For example, a clone instance was changed in Revision 10, but its clone instance was changed in the same way in Revision 11. Barbour *et al.* [2] call this type of changes as the late propagation in clones and point out that in many cases, the late propagation indicates bugs. These cases highlight the importance of our approach, since our synchronized code avoids the late propagation. However, they reduce both precisions and recalls since we consider that manual changes happened in the same revision as the golden standard in our evaluation.
3. CCSync inherits the limitations of the matching algorithm (*i.e*, LCS) in *MCIDiff*, and the limitation can lead to changes in wrong places. For example, if a left-side clone instance has a construct (*e.g.*, a parenthesis "}") and the right-side clone instance has two consecutive identical constructs (*e.g.*, "}}"), CCSync can mismatch them and cause the wrong synchronization eventually.

C. RQ3: Synchronizing Clone Groups

1) Setup: We collected clone groups that were detected by ConQAT with the same parameters in Section V-A. From the detected clone groups, we excluded the ones which are not modified and the ones whose code fragments are not similar at all. According to the number of clone instances, we classified them into 3 categories: $|\mathbf{CG}| = 3$, $|\mathbf{CG}| = 4$ and $|\mathbf{CG}| \geq 5$, where $|\mathbf{CG}|$ is the size of a clone group. As shown in Table V,

TABLE V
RESULTS OF SYNCHRONIZING CLONE GROUP

|CG|	#CG	#MC	#SC	#∩	P	R	I
3	74	1,053	902	614	68.1%	58.3%	40.5%
4	38	798	710	354	49.9%	44.4%	26.3%
≥5	20	578	492	158	32.1%	27.3%	15.0%
Total	132	2,429	2104	1,126	53.5%	46.4%	32.6%

|CG|: size of clone group; #CG: #clone groups; #∩: #(MC ∩ SC); P: Precision; R: Recall; I: Identical;

Column "#CG" lists the number of clone groups. We repeated the evaluation in Section V-B on the clone groups.

2) Result: Table V shows the results. The last row of Columns P and R shows that in total, the precision and the recall for synchronizing clone groups are 53.5% and 46.4%, respectively. In total, Column I shows that in 32.6% of clone groups MCs and SCs are identical. Compared with Table IV, the precisions and recalls are relatively lower. In particularly, with more clone instances in a clone group, the precisions and recalls become lower.

3) Discussion: We analyzed clone groups whose precisions or recalls are below 50%, and we found the following issues:
1. Synchronization rules can conflict with programmers' intention. This problem affects synchronizing clone pairs, and becomes even more serious with the increasing of clone instances in a clone group. In particular, even when two clone instances of a clone group need to be synchronized, the other clone instances may not. This limitation leads to the low precisions in Table V.
2. Two clone instances of a clone group have some common code constructs, but other clone instances of the clone group do not have such common code constructs. As CCSync does not detect such common code constructs, some changes are neglected, which leads to the low recalls in Table V.

The results for synchronizing clone groups can be improved with better rule inference techniques. In addition, as CCSync allows customizing rules, it is feasible to improve the precisions and recalls with manual customization. In particular, after disabling the actions of several inferred synchronization rules, we improve the precisions to more than 70%.

D. Threats to Validity

The threat to construct validity includes the detected code clones. As code clones are not explicit, we have to use the tool, ConQAT, to detect code clones. Its detected code clones may not represent real development. To reduce the threat, we carefully tuned ConQAT, and the threat could be further reduced by introducing experienced programmers to identify code clones. The threat to external validity includes our selected projects. Although we selected hundreds of revisions for each project in our study, the revision ranges do not reflect the whole picture of projects. In addition, the selected projects might not represent all projects. The threat could be reduced by introducing more projects in future work.

VI. DISCUSSION AND FUTURE WORK

Synchronizing quite different code clones. CCSync contains two major components such as inferring rules and synchronizing code clones with rules. When inferring rules, CCSync compares ASTs for common code structures. Comparing with the state-of-the-art approach [30], CCSync has the potential to handle more different code clones, since its generated multisets have both mapped constructs and unmapped constructs. Furthermore, it is feasible to extend CCSync to infer rules for even more different code clones. For example, in RQ1, we extract co-changes from revisions, and these extracted co-changes are useful to infer rules. With more advanced techniques to infer rules, CCSync can synchronize more different code clones.

Comparing with the state-of-the-art tool. We did not compare CCSync with the state-of-the-art tool [30], since it is not publicly available. To present the benefits of CCSync, we carefully select arguments for ConQAT so that the underlying tool can detect Type III clones for synchronization, while the state-of-the-art tool can synchronize Type I and Type II clones.

VII. RELATED WORK

Our approach is related to the following research fields:

Clone Management. Duala-Ekoko *et al.* [6], [7] introduce a tool called CloneTracker that can provide the developers with the assists to track evolutionary clone groups. It uses CRD, a light-weight clone region descriptor to build the correspondence between the clone groups in the consecutive versions. Nguyen *et al.* [30] introduce a clone management tool called JSync to detect the code clone and help developers to make consistency changes when the create or modify cloned code. Lin *et al.*[23] propose a clone-based and interactive approach to assist the developers to edit and modify copy-paste code. CCSync can synchronize code clones whose structures are quite different, complementing the above approaches. Thus, it further reduces the harmfulness of editing clone clones [35].

Code Translation. It has been a hot research topic to translate code from one form to another. Zhong *et al.* [41] propose an approach that supports translating between Java and C#, and their empirical studies [40], [32] analyze various issues in this research direction. Gokhale *et al.* [11] and Nguyen *et al.* [29] improve their approach from different inputs and more advanced techniques, respectively. Meng *et al.* [27] propose an approach that translates code by examples. These approaches translate code from only one direction. Our approach compliments these approaches, since it is bidirectional and fits the needs of clone synchronization. Zhang *et al.* [38] propose an approach that updates similar code based on a predefined diff file. Their approach can update clones with similar structures, while our approach is able to synchronize code clones in quite different structures.

Code Refactoring. Tsantalis *et. al* [33] conduct an empirical study on the refactorability of software clone, and summarize how refactorability is affected by different clone properties. Kim *et al.* [18] conduct a field study and analyze challenges in refactoring. With the support of our approach, it is feasible to implement a tool that allows programmers to update a piece of refactored code, and to reflect its revisions to the original code. In particular, our approach can be applied to maintain the consistency between code fragment before and

after refactoring. When the programmers selectively undo the refactoring [5], the synchronized code fragment before refactoring can be provided.

VIII. Conclusion

In this paper, we present a rule-directed approach to synchronizing code clones and implement a tool CCSync. CCSync utilizes the generated rules to synchronize most of the clone pairs effectively. Our evaluation has shown that CCSync can play an important role in synchronizing clone pairs and it has precision over 92% and recall over 84%. In particular, more than 76% of our generated revisions are identical with manual revisions. In addition, allowing to tailor the actions manually helps to make a difference between the changes that need not be propagated and the ones that need.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. We also thank Lingxiao Jiang for his invaluable feedbacks on this work. This work is sponsored by the 973 Program in China (No. 2015CB352203), the National Nature Science Foundation of China (No. 61572312, No. 61572313, and No. 61272102), the grant of Science and Technology Commission of Shanghai Municipality (No. 15DZ1100305), and partially supported by JSPS Grant-in-Aid for Scientific Research (A) No. 25240009 in Japan.

References

[1] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proc. CSMR*, pages 81–90, 2007.

[2] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proc. ICSM*, pages 273–282, 2011.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. ICSM*, pages 368–377, 1998.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[5] X. Cheng, Y. Chen, Z. Hu, T. Zan, M. Liu, H. Zhong, and J. Zhao. Supporting selective undo for refactoring. In *Proc. SANER*, pages 13–23, 2016.

[6] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. ICSE*, pages 158–167, 2007.

[7] E. Duala-Ekoko and M. P. Robillard. Clonetracker: tool support for code clone management. In *Proc. ICSE*, pages 843–846, 2008.

[8] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. ICSM*, pages 109–118, 1999.

[9] R. Fanta and V. Rajlich. Removing clones from the code. *Journal of Software Maintenance*, 11(4):223–243, 1999.

[10] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[11] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between apis. In *Proc. ICSE*, pages 82–91, 2013.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *Proc. ICSE*, pages 96–105, 2007.

[13] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proc. ESEC/FSE*, pages 55–64, 2007.

[14] E. Jürgens, F. Deissenboeck, and B. Hummel. Clonedetective - A workbench for clone detection research. In *Proc. ICSE*, pages 603–606, 2009.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

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

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. ESEC/FSE*, pages 187–196, 2005.

[18] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proc. ESEC/FSE*, page 50, 2012.

[19] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1):77–108, 1996.

[20] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. WCRE*, pages 170–178, 2007.

[21] J. Krinke. Is cloned code more stable than non-cloned code? In *Proc. SCAM*, pages 57–66, 2008.

[22] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. ICSM*, pages 314–321, 1997.

[23] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao. Clone-based and interactive recommendation for modifying pasted code. In *Proc. ESEC/FSE*, pages 520–531, 2015.

[24] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao. Detecting and summarizing differences across multiple instances of code clones. In *Proc. ICSE*, pages 164–174, 2014.

[25] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proc. ICSME*, pages 227–236, 2008.

[26] A. Lozano and M. Wermelinger. Tracking clones' imprint. In *IWSC*, pages 65–72, 2010.

[27] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proc. ICSE*, pages 502–511, 2013.

[28] M. Mondal, C. K. Roy, and K. A. Schneider. A fine-grained analysis on the evolutionary coupling of cloned code. In *Proc. ICSME*, pages 51–60, 2014.

[29] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining api usage mappings for code migration. In *Proc. ASE*, pages 457–468, 2014.

[30] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2012.

[31] D. C. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proc. 29th ICSE*, pages 116–126, 2007.

[32] L. Shi, H. Zhong, T. Xie, and M. Li. An empirical study on evolution of API documentation. In *Proc. ETAPS/FASE*, pages 416–431. 2011.

[33] N. Tsantalis, D. Mazinanian, and G. P. Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, 2015.

[34] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *Proc. ESEC/FSE*, pages 455–465, 2013.

[35] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can I clone this piece of code here? In *Proc. ASE*, pages 170–179, 2012.

[36] G. Zhang, X. Peng, Z. Xing, S. Jiang, H. Wang, and W. Zhao. Towards contextual and on-demand code clone management by continuous monitoring. In *Proc. ASE*, pages 497–507, 2013.

[37] G. Zhang, X. Peng, Z. Xing, and W. Zhao. Cloning practices: Why developers clone and what can be changed. In *Proc. ICSM*, pages 285–294, 2012.

[38] T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *Proc. ICSE*, 2015.

[39] Y. Zhang, H. A. Basit, S. Jarzabek, D. Anh, and M. Low. Query-based filtering and graphical view generation for clone analysis. In *Proc. ICSM*, pages 376–385, 2008.

[40] H. Zhong, S. Thummalapenta, and T. Xie. Exposing behavioral differences in cross-language API mapping relations. In *Proc. ETAPS/FASE*, pages 130–145, 2013.

[41] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. ICSE*, pages 195–204, 2010.