Almost Rerere: Learning to resolve conflicts in distributed projects

Sergio Luis Herrera Gonzalez and Piero Fraternali
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Piazza Leonardo da Vinci 32, Milan, 20133, Italy.

Abstract—The concurrent development of applications requires reconciling conflicting code updates by different developers. Recent research on the nature of merge conflicts in open source projects shows that a significant fraction of merge conflicts have limited size (one or two lines of code) and are resolved with simple strategies that use code present in the merged versions. Thus the opportunity arises of supporting the resolution of merge conflicts automatically by learning the way in which developers fix them. In this paper we propose a framework for automating the resolution of merge conflicts which learns from the resolutions made by developers and encodes such knowledge into conflict resolution rules applicable to conflicts not seen before. The proposed approach is text-based, does not depend on the programming languages of the merged files and exploits a well-known and general language (search and replacement regular expressions) to encode the conflict resolution rules. Evaluation results on 14,872 conflicts from 25 projects show that the system can synthesize a resolution for ≈ 49% of the conflicts occurred during the merge process (≈ 89% if one considers conflicts that have at least one similar conflict in the data set) and can reproduce exactly the same solution that human developers have applied in ≈ 55% of the cases (≈ 62% for single line conflicts).

Index Terms—Automatic Conflict Resolution, GIT, Code integration.

1 Introduction

The development of large and complex software applications requires the distribution of programming among multiple developers. When the same code base is changed by different actors, inconsistencies may arise between the concurrent changes. This occurrence is called *conflict* [1]. To support distributed development, Version Control Systems (VCS) [2] offer functions to share code, track changes, and identify conflicts caused by the merge of concurrent versions. When conflicts are signalled by the VCS, the developer has the responsibility of resolving them manually, which makes code integration a time-consuming task [3]. Conflict resolution is also repetitive because similar or identical conflicts appear at every iteration. A study of software projects [4] found that repetitiveness of small changes can be very high (up to 70%) and may lead to addressing many similar conflicts during integration again and again.

A recent investigation of a large number (2.731) of open source projects permitted an accurate characterization of conflicts [5]. One of the findings is that over half of the conflicting chunks of code are limited in size (5 lines or less) and the median size of conflicting chunks across all the reviewed projects is 2.0 and 2.5 LOCs for the two merged versions. The analysis also reveals that frequently (in 87% of the cases) the conflict is resolved by exploiting code that is already present in one of the two merged versions or in both. This is confirmed also by a subsequent empirical study of the distributed development of the Microsoft Edge software, which reports that conflicts occur in the majority of merges (80.4%), small conflicting chunks (1 or 2 LOCs) represent almost 1/3 of the total and the manual resolutions tend to follow repetitive patterns [6]. These findings suggests the possibility of applying data-driven methods to learn the way in which human developers resolve merge conflicts by combining existing code segments.

In this paper we develop a method to let a VCS learn how to resolve new conflicts based on similar conflicts addressed previously. The key idea is to exploit the conflict resolutions implemented by human developers in the past to create rules applicable to future similar conflicts. When the first conflict is resolved manually, the conflicting chunk and the manual resolution are processed to derive a *Conflict Resolution Rule (CRR)*. Then the first *Conflict Cluster (CC)* is created and the rule is associated with it. A CC contains a set of conflicts with similar structure that can be solved in the same way. When another conflict arrives, it is compared to the existing clusters. If its similarity to the conflicts of one or more clusters exceeds a threshold then the CRR of the most similar cluster is used to solve it. Otherwise, the user is asked to provide a resolution and a new cluster is created. In both cases the conflict with the committed solution is added to the (new or matched) cluster and CRR generation is executed to derive the rule associated with the cluster.

The paper addresses the following research questions:

- To what extent can the conflicts created by concurrent submissions be resolved automatically exploiting the knowledge embedded in previously resolved conflicts?
- How similar are the automatically generated and the manually provided conflict resolutions?
- What is the impact of the size of the conflict on the viability of automatic resolution?
- What is the impact of the complexity of the conflict, of the involved language constructs and of the strat-

Manuscript received April 19, 2005; revised August 26, 2015.

egy of manual resolution on the similarity between the manual and the automatic resolutions?
- Which reasons hinder the creation of a CRR able to produce a resolution equivalent to the one manually created by the user?
- Can the automatic resolutions that are dissimilar from the manual ones support the developer and shorten the resolution time?

The contribution of the paper can be summarized as follows:

- We introduce the problem of automating the resolution of similar conflicts in concurrent application development and define the learning framework needed to handle it. The same task has been tackled independently in the recent work [6], with the differences explained in the related work section.
- We apply a *single pass online clustering algorithm* [7] with the Jaro-Winkler string similarity measure [8] to assign incoming conflicts to Conflict Clusters. A CC includes conflicts that may be resolved by the same rule.
- We introduce the idea of using Conflict Resolution Rules (CRRs) in the form of *regular search and replacement expressions* to encode the knowledge of how to fix a conflict. To synthesize the CRRs we adapt the approach of [9] and [10], which exploits genetic programming to build a search and replace regular expression from a set of examples specified as pairs of strings. Our method uses as examples the *before-state* and the *after-state* of the conflicts and produces a CRR that is the best fitted search and replace expression that maps the before states of all the conflicts in the CC into the respective after states.
- We illustrate a reference implementation, called *Almost Rerere*, which extends the functionality of the popular Git VCS [1]. Almost Rerere builds on top of the *Git Rerere* plug-in, which resolves automatically conflicts *identical* to already seen instances and helps developers pre-check partial revisions before integrating a complete revision into the master branch. Almost Rerere can resolve conflicts *similar* to those observed in past iterations and can be used throughout the development process to support the semi-automatic resolution of previously unseen conflicts. It learns more and more precise CRRs as the application development progresses.
- We evaluate the approach with 25 open source projects in four languages. We extract 52,430 conflict chunks from such projects and use 14,872 small size conflicts (up to 6 LOCs) to derive the CRRs. Using the manual resolutions provided by the developers as ground truth, we assess the accuracy of the system. The evaluation results show that the system can synthesize a resolution for $\approx 49\%$ of the conflicts produced during the merge process ($\approx 89\%$ if one considers conflicts that have at least one similar conflict in the data set) and can reproduce exactly the same solution that human developers have applied in $\approx 55\%$ of the cases ($\approx 62\%$ for single line conflicts). We illustrate the learning process with some exemplary cases and discuss the limitations we have observed.

1. https://git-scm.com/

The rest of the paper is organized as follows. Section 2 introduces the preliminary concepts and defines the data-driven approach to the synthesis of conflict resolution rules from conflict instances. Section 3 presents the data set used for the evaluation and responds to the essential research questions of the paper. Section 4 describes the system architecture for the implementation of *Almost Rerere*. Section 5 discusses the threats to validity, with a focus on the limitations that hinder the learning of conflict resolution rules and the extension to other scenarios. Section 6 surveys the related work about the identification of code similarities, the generation of text and code from examples and the study of merge conflicts in distributed projects. It also compares our approach to the few recent research works that aim at the automatic synthesis of conflict resolutions and of bug fixes. Finally, Section 7 provides the conclusions and highlights the envisioned future work.

2 Concepts and Approach

The following definitions capture the essential concepts of concurrent development and conflict resolution.

- **Code-base**: a repository of the artifacts that compose the application.
- **Revision**: a version of an artefact of the code base.
- **Update**: a change affecting a revision, consisting of the insertion, modification or deletion of one or more lines of code.
- **Conflict**: the result of merging two revisions that contain inconsistent updates to the same code. A conflict may comprise multiple chunks corresponding to different portions of the code updated inconsistently by two developers.
- **Conflict Chunk**: a region of code affected by conflicting updates. The conflict chunk contains the two colliding versions (conventionally named *version1* and *version2*) belonging to the two merged revisions. In this paper no semantics is attributed to the version index so that the choice of *version1* and *version2* from a conflict chunk can be considered random.
- **Conflict Resolution**: the set of lines to be inserted into the revision in replacement of the conflict chunk.
- **Conflict Resolution Strategy**: the process adopted by the developer to produce the conflict resolution, chosen among five options [5]. The *Version 1* and the *Version 2* strategies use the code coming from the respective revision; the *Concatenation* strategy concatenates both versions in either order; the *Combination* strategy combines lines of code from both versions without modification; the *New Code* or *Manual* strategy exploits new code possibly mixed with code of the colliding versions; the *None* strategy re-installs the pre-submission version.
- **Conflict Complexity**: a difficulty index determined by the proportion of *straightforward chunks*, resolved using the *Version 1*, *Version 2*, *Concatenation*, or *None*

strategy, and *complex chunks*, resolved with the *Combination* or *Manual* strategy [5].

- **Conflict Cluster**: a set of conflicts with the same pattern. A pattern is a search expression that matches the conflicts of the cluster.
- **Conflict Resolution Rule**: a search and replacement expression that matches a conflict with a given pattern and produces a conflict resolution.

Our approach exploits the knowledge embedded in <*version_x*, *resolution*> pairs to synthesize a CRR that, when applied to conflicts similar to those used to create it, produces a resolution close to the intention of the developer. To strike a balance between CRR specificity and generality, conflicts are grouped into clusters based on a similarity metrics and CRRs are synthesised from all the examples in the cluster. To let the system learn from the manual fixes provided by the developer, the CRR associated with a cluster is recomputed when a new conflict is added to it.

Table 1 illustrates a complete example of the conflict resolution process. A sequence of four conflicts is handled. The first conflict (C1) has no antecedents and thus the resolution committed by the user is exploited to initialize the cluster structure and to build the first CRR. The synthesized rule (CRR1.1) has quite a rigid pattern: it replaces whatever comes after `",t"` up to the end of the string delimited by `");"` with the fixed string `", true);"`. The second conflict C2 is found to have high similarity to the cluster (score = 0.9399) and thus CRR1.1 is used to create the resolution. The fix produced by CRR1.1 is proposed to the developer who accepts and commits it (indeed the automatic fix is identical to the ground truth resolution found in the project repository). C2 with its resolution is added to the cluster and the CRR is regenerated. The resulting CRR1.2 now has a slightly more general pattern: it searches for the substring `"rue"` followed by a comma and a word starting with `'t'` and replaces it with the characters of the matched word after the `'t'`. Note that in the regex the matched substring to be used as a replacement is denoted by a so-called *matching group*, which is a sub-expression enclosed within parentheses `(..)`. In practice, the CRR has learnt to find two comma-separated occurrences of `"true"` and to replace them with only one occurrence. When conflict C3 arrives, it is found similar to the cluster (score = 0.9471). Thus CRR1.2 is applied to it and an automatic fix is computed. CRR1.2 does not match any portion of the input and thus the conflict version matching the cluster is proposed as the resolution to the developer. This time the automatic fix is not correct and thus the developer rejects it and commits a different one. The committed fix is used as the correct after state and the conflict C3 with its resolution is added to the cluster, which triggers the re-computation of the CRR. Thanks to the new knowledge acquired from the manual fix of C3 now CRR1.3 has a more general structure. It searches for a first capturing group formed by a sequence of characters and for a second capturing group containing the characters `");"`, the two being separated by a comma, a space and a sequence of characters. It replaces the matched substring with only the content of the capturing groups: in practice, the rule has learnt to delete the last parameter, whatever its name is. Finally, conflict C4 arrives. Again it is found similar to the cluster (score = 0.9573). The rule CRR1.3 is applied to compute the automatic resolution (i.e., by deleting the last parameter). The automatic fix is submitted to the developer, who accepts and commits it (again, we can see that the automatic solution is identical to the GT fix in the project repository). The conflict and the committed resolution are added to the cluster, which triggers the re-creation of the CRR. The new rule, CRR1.4, is identical to the previous one CRR1.3, which means that the algorithm has produced a reusable rule. Obviously, the future addition of other similar conflicts to the cluster may lead to a revision of the CRR so as to adapt it to the new examples.

Note that during conflict-to-cluster matching and cluster creation and extension we always employ one of the two colliding versions, namely V1. However, since the assignment of an index to a version is arbitrary, the choice is random. The same version used for matching is exploited for creating the automatic resolution and for extending the cluster.

Algorithm 1 formalizes the process of CRR synthesis as an online loop that takes in input one conflict chunk at a time and produces in output a conflict resolution, either manually provided by the developer or synthesized by the system. The algorithm relies on two main procedures: *findSimilarCluster*, which compares an incoming conflict to those seen in the past to identify the cluster of most similar conflicts; and *synthesizeRule*, which produces a CRR from the set of conflicts in the cluster.

Algorithm 1: Resolution algorithm

```
Input: c_i the next conflict;
v_i a randomly chosen version from c_i;
Output: s_ci the solution of conflict c_i;
State: CC: current set of clusters (initially empty);
Loop
    cl = findSimilarCluster(CC, < v_i, null >);
    if cl == null then
        /* no similar conflict(s) seen in the past,
         create new cluster */
        /* developer provides manual resolution */
        s_ci = askUser(< c_i, null >);
        newc= createCluster(< v_i, s_ci >);
        newRule= synthesizeRule(newc);
        associateRuleToCluster(newc, newRule);
        CC.addCluster(newc);
    else
        /* cluster with similar conflict(s) exists, use its
         CRR and extend the most similar cluster */
        existingRule = cl.getRule();
        ŝ_ci = applyRule(v_i, existingRule);
        /* developer may accept or ignore the
         proposed automatic resolution */
        s_ci = askUser(< c_i, ŝ_ci >);
        cl = findSimilarCluster(CC, < v_i, s_ci >);
        addConflictToCluster(cl, < v_i, s_ci >);
        updatedRule = synthesizeRule(cl);
        associateRuleToCluster(cl, updatedRule);
    end
    output (s_ci);
EndLoop
```

▶ Conflict C1 arrives.

V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true, true);`
V2	`final Type promote = AnalyzerCaster.promoteNumeric(definition, left.actual, right.actual, true);`

No cluster exists, so no resolution can be computed and the fix committed by the developer (Man res) is used.

Man res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true);`

A new conflict cluster CC1 is created from the conflict and its resolution and a CRR is synthesised for it.

Conflict cluster CC1	**CRR1.1 := regex: `",\st(.*)\);"` replacement: `", true);"`**
V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true);`

The CRR1.1 rule searches for a sequence of characters, starting with a comma followed by a space, by the letter `t` and by any subsequent character until a closing parenthesis and a semicolon, and replaces it with the literal string `', true);'`

▶ Conflict C2 arrives.

V1	`final Type promote = AnalyzerCaster.promoteNumeric(child.actual, true, true);`
V2	`final Type promote = AnalyzerCaster.promoteNumeric(definition, child.actual, true);`

V1 in conflict C2 is similar to the conflicts of cluster C1 (similarity = 0.9399) and thus CRR1 is applied to solve it.
The regex matches the substring highlighted in yellow and replaces it with `", true);"`, which yields the following automatic resolution (Aut res).

Aut res	`final Type promote = AnalyzerCaster.promoteNumeric(child.actual, true);`

The automatic resolution is accepted and committed by the developer. This coincides with the ground truth fix found in the repository (GT res).

GT res	`final Type promote = AnalyzerCaster.promoteNumeric(child.actual, true);`

Conflict C2 is added with its resolution to cluster C1 and the CRR is regenerated.

Conflict cluster CC1	**CRR1.2 := regex: `"rue, t(\w++)"` replacement: `"$1"`**
V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true);`
V1	`final Type promote = AnalyzerCaster.promoteNumeric(child.actual, true, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(child.actual, true);`

The CRR1.2 rule searches for an occurrence of the substring `rue` followed by a comma and a word starting with the character `t` and replaces it with the characters of the matched word after the `t`. If two comma separated occurrences of `true` are found, one is deleted.

▶ Conflict C3 arrives.

V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, false, true);`
V2	`final Type promote = AnalyzerCaster.promoteNumeric(definition, left.actual, false);`

V1 of C3 is similar to CC1 (similarity = 0.9471) and thus CRR1.2 is applied to solve it

Auto res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, false, true);`

This time the automatic resolution is wrong, because CRR1.2 does not match any part of the conflict and thus outputs it as is. As a consequence the automatic resolution is discarded by the developer who commits the correct fix.

GT res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, false);`

Conflict C3 with the developer's fix is added to CC1 and the CRR is regenerated.

Conflict cluster CC1	**CRR1.3 := regex: `"(\w++),\s\w++(\);)"` replacement: `"$1$2"`**
V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true);`
V1	`final Type promote = AnalyzerCaster.promoteNumeric(child.actual, true, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(child.actual, true);`
V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, false, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, false);`

CRR1.3 searches for a first capturing group formed by any sequence of characters, followed by a comma a space and another characters sequence and then for a second capturing group containing the closing character sequence `");"`.
It replaces the searched substring with only the content of the capturing groups: in practice, the rule has learnt to delete the last parameter.

▶ Conflict C4 arrives.

V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, false, true);`
V2	`final Type promote = AnalyzerCaster.promoteNumeric(definition, left.actual, right.actual, false);`

V1 in conflict C4 is similar to the conflicts of cluster C1 (similarity = 0.9573) and thus CRR1.3 is applied to solve it.
The regex matches the substring highlighted in yellow and replaces it with the green one, which yields the following automatic resolution

Auto res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, false);`

The automatic resolution is accepted and committed by the developer (it coincides with the GT resolution in the repository).

GT res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, false);`

Conflict C4 is added to cluster C1 and the CRR is regenerated.

Conflict cluster CC1	**CRR1.4 := regex: `"(\w++),\s\w++(\);)"` replacement: `"$1$2"`**
V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, true);`
V1	`final Type promote = AnalyzerCaster.promoteNumeric(child.actual, true, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(child.actual, true);`
V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, false, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, false);`
V1	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, false, true);`
Res	`final Type promote = AnalyzerCaster.promoteNumeric(left.actual, right.actual, false);`

The CRR1.4 is the same as CRR1.3, i.e., C4 has not improved the knowledge in the cluster.

Table 1: A complete example of the conflict resolution process. In the automatically resolved conflicts C2 and C4, the yellow text shows the matching region and the green one shows the replacement in the resolution.

2.1 Similarity metrics

In a text-based approach the similarity between code chunks can be computed with a string similarity measure. Such a function impacts the formation of the CCs and thus must be chosen with care. The strings to be compared comprise both non-semantic differences (spacing, comments, etc.) and semantic ones (changes in variable names or literals, in the signature or in the body of functions, in directives etc.). The syntax varies based on the programming language. Several string similarity metrics were evaluated. To this end, a test data set was created by extracting 150 strings from the conflicts of open source repositories and by manually grouping them by similarity. The strings included samples of Java, JavaScript and HTML code. The evaluation consisted in extracting one random element at a time from the data set and assigning it to the group with the maximum average similarity under each distance function. An assignment is considered positive if the sample is mapped to the group manually associated to it, negative otherwise. The performance of each distance function is assessed with the standard metrics of precision, recall and F_1 score. The evaluation compared 10 different algorithms from 3 different string similarity approaches: edit distance based (Hamming, Levenshtein, Damerau-Levenshtein, Jaro and Jaro-Winkler), token based (Cosine, Jaccard and Sorensen-Dice) and sequence based (Longest Common Subsequence and Longest Common Substring).

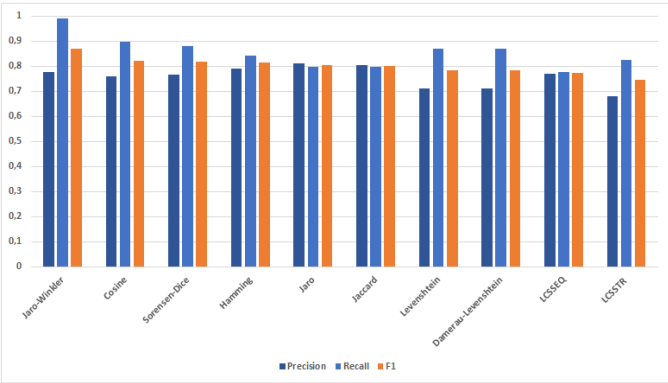

Fig. 1: Evaluation of the string similarity algorithms on the test data set of code samples

Figure 1 reports the precision, recall and F_1 score values computed for each distance function. The *Jaro-Winkler* similarity algorithm has the best performance and was selected for the implementation of the clustering algorithm. A parameter search in the 0-1 interval with step 0.1 was conducted to set the value of the boost parameter of the Jaro-Winkler distance [11]. Figure 1 reports only the performance of the Jaro-Winkler distance with the best parameter value (0.7). The intuition behind the superiority of the Jaro-Winkler metrics is that it gives more importance to differences near the start of the string than to those near the end. It is common in many programming languages that the beginning of a line of code comprises significant reserved words, e.g., type declarations (*int, double, String*), access declarations (*public, private, protected*), flow control specifications (*if, while, switch*), etc. that determine the semantics of the segment and are likely to remain unchanged. The end of a code line, on the other hand, is occupied by variables and operations declared by the developer, which are semantically weaker and more likely to vary.

2.2 Conflict clustering

In Algorithm 1 the assignment of conflicts to clusters is managed by the function *findSimilarCluster*, which takes in input one of the two colliding versions of the conflict and returns a matching cluster or *null*. The function takes as an optional input parameter the resolution. If this is omitted it computes the similarity between the input version and the conflict versions of the cluster. Otherwise it computes the similarity between the conflict version *and its resolution* and all the version+resolution pairs of the cluster. The similarity between version+resolution pairs is the average of the version similarity and of the resolution similarity. The similarity to the multiple elements of a cluster is the average distance to the individual members. The min and max functions were also tested and the average function was retained as the one yielding the best performances.

Function *findSimilarCluster* is used to implement an online one pass clustering algorithm [7] that creates the clustering structure by processing one item at a time with no memory of the preceding assignments. When a conflict is received, the cluster with highest similarity score is searched:

- If the score of the most similar cluster is below a threshold, a null value is returned, which causes a new cluster to be generated and the conflict to be assigned to it.
- Otherwise, the most similar cluster is returned.

The value of the threshold for cluster creation is set to 0.80. This value was determined by repeating the string-to-group assignment test used for selecting the similarity function with the Jaro-Winkler function and threshold values ranging from .1 to 1. The a posteriori validation of both the similarity metrics and of the threshold value proved that the choices made yield the best accuracy of the automatically generated resolutions.

Note that function *findSimilarCluster* is invoked also when the committed fix (automatically or manually created) becomes known, so as to find the cluster most similar to the version+resolution pair and assign the current conflict to it.

2.3 CRR generation

As shown in Algorithm 1, the generation of the CRR of a cluster is triggered by the commit of a conflict resolution, represented by the function *askUser*. The resolution can be either the one proposed by Almost Rerere and accepted by the developer or the one programmed manually by the developer, if no cluster of similar conflicts is found or the automatic fix suggested by the system does not meet the developer's expectation.

After the addition of a conflict to a cluster, procedure *synthesizeRule* creates a new CRR or an improved version of an existing CRR from the content of the cluster. The synthesis of CRRs exploits the general-purpose string search and replacement algorithm of [10], which takes as input the pairs describing the original string and the desired modified string and outputs a search pattern and a replacement

expression. The former is a regular expression (regex) that describes both the portions of the string to be replaced and those to be reused; the latter describes how to build the modified string. The method of [10] employs a Genetic Programming algorithm inspired by concepts of biological evolution such as reproduction, mutation, recombination, and selection. The best regular expression is chosen based on a fitness function. The set of examples is divided in three subsets: training, testing and validation. The training examples are used to generate an initial population of 16 candidate expressions for each training sample. The validation set is used to measure the fitness of the candidates in the initial population. The candidate expressions are applied to the test samples and the precision and recall with respect to the ground truth are computed, as well as the expression complexity. Next, the best candidates are selected and recombined in the next iteration of the process. Finally, the test set is used to evaluate the best candidate expression. When the number of available samples is small, the algorithm is sensitive to the order in which the samples are split into the training and the validation sets. To mitigate this problem, the samples are randomly assigned to the training, testing and validation sets and the algorithm is executed multiple times. If the generated CCR is the same across the executions, which indicates that the algorithm has converged, it is saved. Otherwise, all solutions are kept, and the CCR is composed as the disjunction of the computed expressions. In the experiments, two rounds of executions ensured the best trade-off between performance and accuracy of the synthesised CRR.

The method of [10] has been adapted to take as input a conflict cluster, to dynamically partition the input samples into the training, validation and testing sets, and to output a CRR for each cluster. For the generation of the CRR, the parameters of the genetic algorithm were tuned as follows: the number of evolution cycles was set to 30 and the number of execution threads was set to 2[2].

3 Evaluation

The quantitative evaluation of the CRR synthesis focuses on the following aspects:

- The percentage of the merge conflicts that can be resolved automatically (Question 1).
- The identity of the automatically generated resolutions to those manually provided by the developer, i.e., the accuracy of the automatic resolutions. Accuracy is defined as [number of identical resolutions] / [number of computed resolutions] (Question 2).
- The impact of the size of the conflict on the accuracy of the automatically generated resolutions (Question 3).
- The impact of the strategy of manual resolution (V1, V2, CC, NC) and of the language constructs on the automatic resolutions (Question 4).
- The similarity of the non-identical resolutions to those manually provided by the developer (Question 5).

2. Detailed information about the algorithm parameters can be found in the GitHub documentation of the original project https://github.com/MaLeLabTs/RegexGenerator.

3.1 Evaluation process

The evaluation workflow is organized as follows:

1) The conflict data set is created by reproducing the commits in the Git repositories of the chosen projects, with the technique of [5]. For each conflict, the conflict chunk and the manual resolution are extracted. The manual resolution is used as *ground truth* to assess the identity or similarity of the automatic and manual resolutions. Only conflicts chunks up to 6 LOCs are retained.
2) For each project, the first conflict is extracted. The conflict and its manual resolution are exploited to initialize the clustering procedure of Algorithm 1. This yields the first CC and its associated CCR. The conflict is counted as a failure because no automatic resolution can be computed for it and its metadata (size, complexity, manual resolution strategy) are recorded for evaluation purposes. For Java projects the programming constructs are extracted and recorded too.
3) Then the successive conflicts are processed according to Algorithm 1. If the conflict can be resolved automatically by a CRR the distance between the automatic and the manual resolution is recorded for evaluation. If the conflict cannot be resolved (i.e., it is assigned to a new cluster), it is counted as a failure. The conflict metadata are saved in both cases.

3.2 Data set

The data set used to evaluate the synthesis of automatic resolutions consists of conflict chunks extracted from the submission logs of distributed projects hosted in public repositories. The first 5 projects were selected from the manual analysis of [5] and other 13 Java projects were taken from their automatic analysis data set. We chose the projects with the highest number of conflicting merges and discarded projects that were no longer available due to migration to other repositories, etc. Seven additional non-Java projects in JavaScript, PHP, and Phyton were selected by mining the most popular and trending projects in Github[3], the Linux foundation[4], the open JS foundation[5] and LibHunt[6]. We took those with the highest number of conflicting merges. Note that one cannot know in advance how many conflicts a project will produce when replaying the merge logs. It often happens that very large projects are logged in the repository in such a way that the replay produces few or even no conflicts (e.g., due to the use of re-basing before committing). The selection procedure exploited a try and discard approach whereby we downloaded each project in turn, replayed its commit history and retained it only if more than 100 conflicts were found. For demonstrating language independence the non-Java conflicts account for 28.85% of the total number of conflicts.

Table 2 reports the essential statistics. In total the data set contains 25 projects from which we identified 9,537 failed

3. https://github.com/trending
4. https://www.linuxfoundation.org/
5. https://openjsf.org/
6. https://www.libhunt.com/

merges and mined 52,340 conflict chunks. Based on the observation of [5] and [6] that a large fraction of conflicts have small size, from the total number of conflict chunks we extracted 14,872 samples of up to 6 LOCs (up to 3 LOCs per version). This filter yielded the conflict chunks used to evaluate our approach.

Project	Commits	Merges	Devs	Failed Merges	Conflict Chunks
Java					
Mct	1062	221	14	17	57
Twitter4J	2340	316	123	59	139
Lombok	3098	249	98	47	157
Voldemort	4959	546	59	68	724
Antlr4	8493	1881	233	388	2973
Android Arabic Reader	12586	4267	29	701	1556
Atlas	13631	2959	39	537	2216
Wro4j	4567	1485	34	395	2571
Eucalyptus	26760	6893	42	602	3892
Fred	32308	1375	30	258	1824
Nuxeo	118211	8302	30	526	2292
Universal Media Server	9945	1397	30	357	2096
Zanta Server	13102	2823	17	370	1560
Zk	28461	4001	30	660	4923
Keycloak	14940	4650	337	101	983
Elastic Search	117701	5137	356	564	3329
Realm-java	10418	3406	99	684	2239
Grails-core	21204	3042	224	509	1515
PHP					
Drupal	51707	299	30	143	332
Joomla	52910	7138	30	928	8600
JavaScript					
Pdf.js	14553	5627	30	146	384
Spectrum	14280	4547	94	361	1126
Webpack	14979	4343	715	312	3938
Python					
Sentry	41148	2407	403	255	1048
Hyperspy	14075	3318	59	549	1956
Total	**647438**	**80629**	**3185**	**9537**	**52430**

Table 2: The projects used in the evaluation and the data set statistics

Table 3 shows the incidence of single (SL) and multi-line (ML) conflicts in the data set. SL conflicts consist of 2 LOCs, one for each version. ML conflicts contain up to 6 LOCs, divided into the two versions. The proportion of small size ($\leq$ 6 LOCs) conflicts with respect to the total is $\approx 30\%$, which confirms the importance of small size conflicts already reported by [5] and [6].

3.3 Automatically solved conflicts (Question 1)

Table 3 shows the absolute number and the percentage of conflicts for which an automatic resolution is synthesized. Overall an automatic resolution is computed for 48,81% of the conflicts, with the minimum value for the Mct project (13.33%), which features a small percentage of conflicts with less than 6 LOCs, and the maximum value for the Drupal project (95.38%). The size of the conflict chunk affects the ability to compute an automatic resolution: 58,85% of the SL conflicts admit an automatic resolution vs 40,96% of the ML conflicts. This result is affected by the presence of *singleton conflicts* (SC), i.e., conflicts having a structure that appears only once in the merge history of the project. SCs are always allocated to a singleton cluster because no other conflict with a sufficiently similar content is found. Thus the system cannot learn a resolution for this class. If the singleton conflicts are ignored, the percentage of automatic resolutions for SL and ML conflicts jumps to 88,66% *for the conflicts that have at least one similar conflict in the data set*.

Project	Conflicts			Solved conflicts		
	Total	SL	ML	Total	SL	ML
Mct	15	5	10	2 (13.33%)	0 (0.00%)	2 (20.00%)
Twitter4J	41	22	19	8 (19.51%)	6 (27.27%)	2 (10.53%)
Lombok	89	37	52	32 (35.96%)	11 (29.73%)	21 (40.38%)
Voldemort	312	168	144	180 (57.69%)	124 (73.81%)	56 (38.89%)
Antlr4	925	311	614	465 (50.27%)	186 (59.81%)	279 (45.44%)
Android Arabic Reader	421	179	242	160 (38.00%)	70 (39.11%)	90 (37.19%)
Atlas	1025	473	552	438 (42.73%)	234 (49.47%)	204 (36.96%)
Wro4j	1072	276	796	695 (64.83%)	184 (66.67%)	511 (64.20%)
Eucalyptus	667	266	401	294 (44.08%)	121 (45.49%)	173 (43.14%)
Fred	848	330	518	370 (43.63%)	150 (45.45%)	220 (42.47%)
Nuxeo	372	138	234	116 (31.18%)	57 (41.30%)	59 (25.21%)
Universal Media Server	477	208	269	161 (33.75%)	92 (44.23%)	69 (25.65%)
Zanta Server	408	156	252	112 (27.45%)	48 (30.77%)	64 (22.40%)
Zk	906	655	251	624 (68.87%)	549 (83.82%)	75 (29.88%)
Keycloak	144	53	91	66 (45.83%)	27 (50.94%)	39 (42.86%)
Elastic Search	1927	850	1077	1110 (57.60%)	595 (70.00%)	515 (47.82%)
Realm-java	625	272	353	265 (42.40%)	151 (55.51%)	114 (32.29%)
Grails-core	307	201	106	177 (57.65%)	140 (69.65%)	37 (34.91%)
Drupal	65	64	1	62 (95.38%)	62 (96.88%)	0 (0.00%)
Joomla	2220	1104	1116	1312 (59.10%)	755 (68.39%)	557 (49.91%)
Pdf.js	115	39	76	21 (18.26%)	12 (30.77%)	9 (11.84%)
Spectrum	343	124	219	84 (24.49%)	35 (28.23%)	49 (22.37%)
Webpack	452	181	271	147 (32.52%)	73 (40.33%)	74 (27.31%)
Sentry	269	100	169	83 (30.86%)	38 (38.00%)	45 (26.63%)
Hyperspy	827	317	510	275 (33.25%)	122 (38.49%)	153 (30.00%)
Total	**14872**	**6529**	**8343**	**7259 (48.81%)**	**3842 (58.85%)**	**3417 (40.96%)**

Table 3: Incidence of single line (SL) and multi-line (ML) conflicts and number and percentage of conflicts for which an automatic resolution is synthesized

3.4 Accuracy (Question 2 and 3)

Table 4 shows the accuracy of the automatic resolution, defined as (number of identical resolutions) / (number of computed resolutions). Overall the automatic synthesis reaches an accuracy of 54,86%, which means that in more than half of the cases when the system computes an automatic resolution this is identical to the one provided by the developer. As expected, the system has more difficulty in resolving larger conflicts: accuracy is 62.29% for single line conflicts and drops to 46.50% for multi-line conflicts.

Note that Algorithm 1 is non-deterministic. When a conflict arrives one of the versions is picked at random to search for a similar cluster. However, when a manually resolved conflict is added to a cluster, both the before-state with the matched version (say V1) and the after-state with the developer's resolution become known to the system. The after-state implicitly contains information also about the unseen colliding version (say V2). The resolution provided by the developer may contain code not present in the before-state (V1), either taken from the other version (V2) or added anew in the resolution. This knowledge is exploited to produce CRRs mimicking the human fix. To assess the impact of non-determinism on accuracy, we have contrasted four alternative configurations of Algorithm 1: 1) the approach in which only one of the two colliding versions (say V1) is chosen randomly, used to build a pair (before-state, after-state) and added to the cluster for deriving the CRR. 2) An approach in which both colliding versions (V1

Project	SL	ML
MCT	0 (0.00%)	2 (100.00%)
Twitter4J	5 (83.33%)	1 (50.00%)
Lombok	10 (90.91%)	9 (42.86%)
Voldemort	90 (72.58%)	31 (55.36%)
Antlr4	96 (51.61%)	121 (43.37%)
Android Arabic Reader	30 (42.86%)	44 (48.89%)
Atlas	116 (49.57%)	66 (32.35%)
Wro4j	69 (37.50%)	171 (33.46%)
Eucalyptus	67 (55.37%)	49 (28.32%)
Fred	93 (62.00%)	134 (60.91%)
Nuxeo	21 (36.84%)	28 (47.46%)
Universal Media Server	54 (58.70%)	33 (47.83%)
Zanata-server	17 (35.42%)	23 (35.94%)
Zk	484 (88.16%)	52 (69.33%)
Keycloak	16 (59.26%)	17 (43.59%)
Elastic Search	263 (44.20%)	185 (35.92%)
Realm-java	59 (39.07%)	55 (48.25%)
Grails-core	131 (93.57%)	26 (70.27%)
Drupal	42 (67.74%)	0 (0.00%)
Joomla	550 (72.85%)	372 (66.79%)
Pdf.js	10 (83.33%)	8 (88.89%)
Spectrum	17 (48.57%)	22 (44.90%)
Webpack	55 (75.34%)	51 (68.92%)
Sentry	24 (63.16%)	22 (48.89%)
Hyperspy	74 (60.66%)	67 (43.79%)
Total	**2393**	**1589**
Accuracy	**62.29%**	**46.50%**

Table 4: Conflicts for which the automatic resolution is identical to the manual one. Accuracy is defined as (number of identical resolutions) / (number of computed resolutions)

and V2) are used to add a pair (V1, after-state) and (V2, after-state) to the cluster before regenerating the CRR. 3) An approach in which the two versions are concatenated and a single pair (V1+V2, after-state) is added to the cluster. 4) An approach in which only one version is used, but not picked at random. The chosen version is the one that has the highest similarity to a cluster. Option 2, 3 and 4 did not yield an accuracy improvement but augmented the noise, i.e., the number of conflicts fixed in a way dissimilar from the developer's resolution.

3.5 Resolution Strategy (Question 4)

The complexity of the conflicts, as defined by the resolution strategy applied by the developer [5], is another factor that may affect the synthesis of automatic resolutions. The intuition is that if the manual resolution adopts a simple strategy that selects or concatenates code already present in the submitted versions (strategy V1, V2, CC) learning this type of resolution strategy should be easier. Table 5 shows the number and percentage of solved/identically solved conflicts and the accuracy of resolution for the strategies V1, V2, CC, and NC. The V1 and V2 strategies represent the largest class both in terms of their contribution to the automatic resolutions (63,29%) and in terms of the accuracy of the automatic fix (71,68%), followed by the NC strategy (contribution 31,02%, accuracy 29,48%). The most problematic strategy to capture is CC (contribution 5,69%, accuracy 6,05%). The relative difficulty of capturing the strategies is reflected not only in the capacity of producing the automatic fix but also in the ability of creating a fix equal to the manual one. As shown in Table 5 accuracy is maximal for V1/V2 and minimal for CC. Note that the counter-intuitive result that the "simple" strategy CC yields worse results than the NC one is explained by the fact that the CC resolutions are the least represented in the data set and thus Almost Rerere does not learn their patterns well.

	Tot.	Solved	Ident.	Solved %	Solved contr.	Acc.
SL conflicts						
V1|V2	4748	2829	2104	73.63%	87.92%	74.37%
CC	398	248	11	6.46%	0.46%	4.44%
NC	1383	765	278	19.91%	11.62%	36.34%
	6529	3842	2393			
ML conflicts						
V1|V2	4460	1765	1189	51.65%	74.83%	67.37%
CC	394	165	14	4.83%	0.88%	8.48%
NC	3489	1487	386	43.52%	24.29%	25.96%
	8343	3417	1589			
SL + ML conflicts						
V1|V2	9208	4594	3293	63.29%	82.70%	71.68%
CC	792	413	25	5.69%	0.63%	6.05%
NC	4872	2252	664	31.02%	16.67%	29.48%
	14872	7259	3982			

Table 5: Resolved and identically resolved conflicts by strategy. CC = code concatenation; NC = code combination or new code.

The detailed breakdown of the data regarding the strategy analysis per project can be found in the Appendix: Table 14 reports the split for single line conflicts and Table 15 for multi-line conflicts.

3.6 Language constructs (Question 4)

Table 6 shows the number and the percentage of the solved conflicts and of the identically solved conflicts for the main language constructs. The statistics are computed with the software of [5] and apply only to the Java projects. The analysis confirms the findings of [6] which underlines the relevance of the directive section of the program for the synthesis of conflict resolutions based on repetitive patterns. In that work, the generation of resolutions is limited to the include and macro directives of C++ which account for 12.34% of all the conflicts and can be managed accurately by means of program synthesis. In our work the synthesis of resolutions is applied to all classes of constructs and the conflicts involving the Java import clause account for 20,38% of the automatically resolved conflicts and for 9,84% of the identically resolved conflicts with 24,43% accuracy. The category of statements in which our approach shows the best results is method invocation, which accounts for 23,82% of the resolved conflicts and for 27,70% of identically resolved conflicts, with 52,47% accuracy.

3.7 Similarity (Question 5)

The last question is whether an automatic resolution can be useful even if not identical to that provided by the manual developer. The rationale is that a sufficiently similar resolution can be semantically equivalent to the manual one or at least usable as a hint for speeding up the creation of the manual resolution. To verify the utility of automatic solutions, we classify them according to the Jaro-Winkler similarity with the manual ones. Three intervals are considered: *High* (1 - 0.9) for the which synthesized

Constructs	Resolved	Identically resolved
Method invocation	1698 (23.82%)	891 (24.70%)
Import	1453 (20.38%)	355 (9.84%)
Attribute	890 (12.49%)	603 (16.71%)
Variable	688 (9.65%)	352 (9.76%)
Method signature	650 (9.12%)	401 (11.11%)

Table 6: Resolved and identically resolved conflicts by Java construct. Only constructs accounting for more that 5% of the automatically resolved cases are shown. The percentage refers to the fraction of the resolved or identically resolved conflicts in which the construct appears.

resolution is equal or almost identical to the original one; *Mid* (0.89 - 0.80) for which the synthesized resolution is close to the original one with only small variations; finally *Low* for those with similarity less than 0.79, for which the synthesized resolution is rather different from the original one. The intervals were selected by a systematic review of the resolutions. The generated resolutions were compared to the developer's resolution. If the generated resolution was different from the developer's one, it was classified as follows: with *high similarity* if the code produced the same result, with minor variants, e.g., in the amount of white space or in the comments; with *mid similarity* if the code was syntactically correct but produced a different result, e.g., due to different code annotations, constant values or variable initializers; with *low similarity* otherwise. From such a classification the cutoff values emerged.

Table 7 reports the results of the similarity analysis.

The percentage of high similarity resolved conflicts raises to 66,68% from the value of the accuracy 54,86%, which considers only identical resolved conflicts.

SL conflicts		
High (1-0.9)	**Mid (0.89-0.79)**	**Low ($\leq$ 0.79)**
2860 (74.44%)	408 (10.62%)	574 (14.94%)
ML conflicts		
High (1-0.9)	**Mid (0.89-0.79)**	**Low ($\leq$ 0.79)**
1981(57.97%)	657 (19.23%)	779 (22.80%)
SL & ML conflicts		
High (1-0.9)	**Mid (0.89-0.79)**	**Low ($\leq$ 0.79)**
4841 (66.69%)	1065 (14.67%)	1353 (18.64%)

Table 7: Automatic resolution similarity to the manual resolution

The detailed breakdown of the similarity analysis per project can be found in the Appendix: Table 16 reports the split for single line conflicts and Table 17 for multi-line conflicts.

An example of a high similarity (0.91) automatic resolution equivalent to the manual one is the following case from the Antlr4 project:

```
DFAState D;          // manual
DFAState D = null; // automatic
```

The automatic resolution inserts the null initializer for the variable, following a pattern learned from similar examples in the conflict cluster.

3.8 Examples

In this section we illustrate a few cases of CRR synthesis and application. For space reasons clusters are abbreviated to include only some of their conflicts. To facilitate the interpretation of the CRR, in Tables 8-11 the matching groups in the regex are color-coded and the same colors are used to highlight the portion of the input matched by the regex and the part of the output updated by the replacement expression.

The simplest rule is the one that mimics the decision of the developer to copy the code of one of the two versions. For example, the cluster of Table 8 is built during the merge process of the Voldemort project.

The CRR associated with it is simply `\d?+`, which is a dummy rule that does not match with the input and simply returns it as the output.

Another typical CRR scheme is the one that determines the insertion or the replacement of a token picked from one of the two versions into the resolution. For example, Table 9 shows a conflict cluster built during the merge process of the Antl4 project. The CRR associated with such a cluster contains the selection regular expression `Toke\w`, which matches the occurrence of the word `Token`. Then the replacement expression `Input` replaces the match with the string `Input`. The CRR maps the input:

```
System.out.println("reportContextSensitivity
    decision="
  +dfa.decision+":"
  +acceptState.s0.configset+ ", input="
  +parser.getTokenStream().getText(interval));
```

into the output:

```
System.out.println("reportContextSensitivity
    decision="
  +dfa.decision+":"
  +acceptState.s0.configset+ ", input="
  +parser.getInputStream().getText(interval));
```

Such an output is the same as the one created by the developer by combining the two colliding versions with strategy NC.

When the resolution requires modifying the code in multiple places, the learned rule comprises more than one capturing group and determines multiple matches. Table 10 shows a conflict cluster example.

The CRR associated with the cluster comprises two capturing groups, enclosed in parentheses, which match different parts of the input. The replacement expression adds the type parameter section in front of the matched substrings. This CRR transforms the input:

```
public void reportContextSensitivity(
 @NotNull Parser recognizer,
 @NotNull DFA dfa, int startIndex, int
     stopIndex,
 @NotNull SimulatorState acceptState)
```

into the output:

```
public <T extends Symbol> void
    reportContextSensitivity(
 @NotNull Parser<T> recognizer,
```

	CRR regex: "\d?+" Replacement: ""
V1	result = new voldemort.client.protocol.pb.VAdminProto.DeletePartitionEntriesRequest();
Res	result = new voldemort.client.protocol.pb.VAdminProto.DeletePartitionEntriesRequest();
V1	result = new voldemort.client.protocol.pb.VAdminProto.DeletePartitionEntriesResponse();
Res	result = new voldemort.client.protocol.pb.VAdminProto.DeletePartitionEntriesResponse();
V1	result = new voldemort.client.protocol.pb.VAdminProto.AsyncOperationStatusRequest();
Res	result = new voldemort.client.protocol.pb.VAdminProto.AsyncOperationStatusRequest();
V1	result = new voldemort.client.protocol.pb.VAdminProto.AsyncOperationStatusResponse();
Res	result = new voldemort.client.protocol.pb.VAdminProto.AsyncOperationStatusResponse();

Table 8: Example of a cluster that generates a dummy rule which matches nothing and returns the input string without modifications.

	CRR regex: "Toke\w" Replacement: "Input"
V1	System.out.println ("reportAttemptingFullContext decision="+dfa.decision+":"+ initialState.s0.configset+ ", input="+ parser.getTokenStream().getText(interval));
Res	System.out.println ("reportAttemptingFullContext decision="+dfa.decision+":"+ initialState.s0.configset+ ", input="+ parser.getInputStream().getText(interval));
V1	System.out.println ("reportContextSensitivity decision="+dfa.decision+":"+ acceptState.s0.configset+ ", input="+ parser.get Token Stream().getText(interval));
Res	System.out.println ("reportContextSensitivity decision="+dfa.decision+":"+ acceptState.s0.configset+ ", input="+ parser.get Input Stream().getText(interval));

Table 9: Example of a cluster with a CRR that requires the replacement of a token. The matching part of the input and the replacement in the output are represented in red.

CRR regex: "(\w++\s\w++\(@\w++\s\w++) (\s\w++,\s@\w++\s\w++\s\w++,\s\w++\s\w++,\s\w++\s\w++,\s@\w++\s\w++)" Replacement: "<T extends Symbol> $1<T>$2"	
V1	public void parseAttributeDef(@Nullable Parser recognizer, @NotNull DFA dfa, int startIndex, int stopIndex, @NotNull SimulatorState<T> acceptState)
Res	public <T extends Symbol> void parseAttributeDef(@Nullable Parser<T> recognizer, @NotNull DFA dfa, int startIndex, int stopIndex, @NotNull SimulatorState<T> acceptState)
V1	public void reportContextSensitivity(@NotNull Parser recognizer, @NotNull DFA dfa, int startIndex, int stopIndex, @NotNull SimulatorState<T> acceptState)
Res	public <T extends Symbol> void reportContextSensitivity(@NotNull Parser<T> recognizer, @NotNull DFA dfa, int startIndex, int stopIndex, @NotNull SimulatorState<T> acceptState)
V1	public void reportContextSensitivity(@NotNull Parser recognizer, @NotNull DFA dfa, int startIndex, int stopIndex, @NotNull SimulatorState acceptState)
Res	public <T extends Symbol> void reportContextSensitivity(@NotNull Parser <T> recognizer, @NotNull DFA dfa, int startIndex, int stopIndex, @NotNull SimulatorState acceptState)

Table 10: Example of a cluster with a CRR that requires multiple matching regions. The two matching groups and the respective matched input parts and output replacements are represented in different colors.

```
3   @NotNull DFA dfa, int startIndex, int
        stopIndex,
4   @NotNull SimulatorState acceptState)
```

The synthesised resolution exactly reproduces the fix in which the developer manually modified the code of the two input versions using the NC strategy.

As a final example, we show a case in which the CRR combines multiple matches and the exchange of position of substrings within the input. Table 11 shows the conflict cluster. The regex of the CRR contains three capturing groups. It matches three sub-strings and swaps them. When applied to the input:

```
1   "s0-'else'->:s1=>1\n" + "s0-'}'->:s2=>2\n";
```

it produces the output:

```
1   "s0-'}'->:s2=>2\n" + "s0-'else'->:s1=>1\n";
```

The output corresponds to the manual resolution that the developer has created using the NC strategy.

4 Implementation

The approach to conflict resolution described in this paper is implemented in a tool called *Almost Rerere*, which extends *Git Rerere* (**RE**use **RE**corded **RE**solution)[7], a plug-in of the popular Git VCS.

Git Rerere is conceived to resolve conflicts that have already been handled in previous code integration steps. When a new conflict occurs the tool automatically records it in a pre-image file and once the conflict has been resolved manually by the developer it stores the conflict resolution in a post-image file. When exactly the same conflict occurs again the recorded solution is applied to the conflict automatically. Git Rerere only automates the resolution of multiple identical conflicts and cannot handle similar pre-images to apply a recorded solution to a non-identical conflict. *Almost Rerere* aims at resolving automatically not only conflicts that are identical to previously seen instances, but also those that are similar to instances solved in the past. It identifies conflicts similar to each other, groups them into clusters based on a distance metrics, and associates each cluster

7. https://git-scm.com/docs/git-rerere

CRR regex:	**"(\w++'\->:\w\d=>\d)(\\\w"\s\+\s"\w\d\-')(}'\->:\w\d=>\d)"**
Replacement:	**"$3$2$1"**
V1	"c0-'else'->:s1=>1\n" + "c0-'}'->:s4=>1\n";
Res	"c0-'}'->:s4=>1\n" + "c0-'else'->:s1=>1\n";
V1	"s1-'else'->:s3=>1\n" + "s1-'}'->:s2=>2\n";
Res	"s1-'}'->:s2=>2\n" + "s1-'else'->:s3=>1\n";
V1	"s0-' else'->:s1=>1 \n" + "s0-' }'->:s2=>2 \n";
Res	"s0-' }'->:s2=>2 \n" + "s0-' else'->:s1=>1 \n";

Table 11: Example of a cluster with a CRR that requires multiple matches and the exchange of position of the matched substrings. The three matching groups and the respective matched input parts and output replacements are represented in different colors.

with a search and replacement expression synthesized from the manual conflict resolutions of the cluster, applicable for solving future conflicts matching the cluster membership criterion. Figure 2 shows the architecture of Almost Rerere, which comprises four main components: the Submission Manager, the Cluster Manager, the CRR Generator, and the Conflict Resolver.

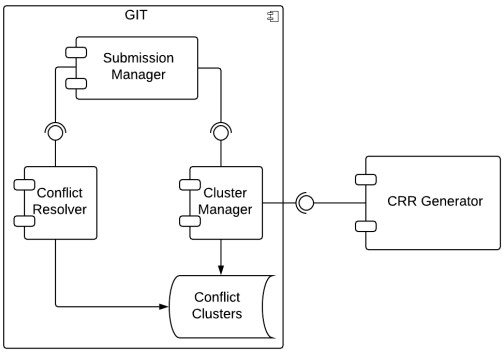

Fig. 2: *Almost Rerere* architecture

The Submission Manager extends *Git Rerere* and orchestrates the processing of a merge or commit command issued by the developer. The Cluster Manager implements the online hierarchical clustering algorithm that assigns an input conflict to an existing or new cluster. The CRR Generator exploits the method for the generation of a search and replacement expressions proposed in [10] and is triggered every time a conflict is added to a cluster. Finally, the Conflict Resolver is called when a new conflict occurs. It searches for the cluster with the highest similarity index to the conflict, extracts the CRR, applies it, and returns the result as the possible solution to the conflict.

Almost Rerere can be used in two phases of the code integration process, represented by the *git merge* and the *git commit* commands. When *git merge* is executed, Git will try to execute an automatic merge between the local and the remote repositories. If conflicts arise, the automatic merge fails, and the conflicts are identified. *Almost Rerere* is invoked, resolves the conflicts by applying the CRRs generated in previous integration cycles, and proposes a solution to the developer. The *git commit* can be executed after all the existing conflicts have been resolved. In this case Almost Rerere collects the conflict resolutions that the user pushed to the repository, calls the Cluster Manager to add the new conflicts, and invokes the CRR generator to update the CRR of the modified clusters. These operations are repeated at every code integration cycle. The former is executed in real-time to return the solution to the developer, while the latter is run in background.

5 Threats to Validity and Reproducibility

5.1 Construction validity

The experiments exploited the conflict extraction software employed in [5] and later in [6] to mine conflicts from GitHub projects. The tool identifies merges with the help of the Git command that lists the commits with more than one parent. Then, the merge process can be replayed and the failed merges can be identified to extract the conflict chunks and the related statistics and manual resolutions. This choice minimizes the occurrence of construction errors and enables the replication of the data set creation procedure. As noted in [12], the resolutions extracted from project merge logs may contain faulty code. For the sake of our study the presence of incorrect code does not hamper the evaluation, which focuses on learning to reproduce the human resolutions. However, in a real scenario one should be aware that the quality of the manual resolutions used to train the system directly impacts the quality of the automatic resolution proposals.

5.2 Internal validity

5.2.1 *Text-based approach*

The proposed approach is text-based and thus independent of the programming language. It exploits only the minimum amount of knowledge: the code of the colliding versions and that of the developer's resolution. It does not require the definition of conflict patterns nor the knowledge of the types of errors to process. It can be applied to projects using any programming and markup language, even when they are mixed in the same artefact.

However, the purely text-based approach overlooks significant semantics. Table 12 illustrates a simple example from the Voldemort project: for a cluster of multiple conflicts the table shows the V1 and V2 versions, the manual resolution of the developer (Man) and the automatic resolution of Almost Rerere (Aut). Even though the manual and automatic resolutions are structurally similar, the former contains a decision that depends on background knowledge not available to Almost Rerere: the choice of adding 1 to the integer value of the argument.

A simple solution to scenarios such as this one is to edit the CRR associated to the CC. The use of a general

V1	`getDescriptor().getMessageTypes().get(44);`
V2	`getDescriptor().getMessageTypes().get(45);`
Man	`getDescriptor().getMessageTypes().get(46);`
Aut	`getDescriptor().getMessageTypes().get(45);`
V1	`getDescriptor().getMessageTypes().get(45);`
V2	`getDescriptor().getMessageTypes().get(46);`
Man	`getDescriptor().getMessageTypes().get(47);`
Aut	`getDescriptor().getMessageTypes().get(46);`
V1	`getDescriptor().getMessageTypes().get(49);`
V2	`getDescriptor().getMessageTypes().get(50);`
Man	`getDescriptor().getMessageTypes().get(51);`
Aut	`getDescriptor().getMessageTypes().get(45);`

Table 12: A conflict resolution in which the developer applied a decision that depends on background knowledge

purpose and well known language such as that of the search and replacement regular expressions facilitates the inclusion of specific semantics in the automatically synthesized rules without the need of learning a domain specific language.

5.2.2 Learning method

Unlike other methods such as [6], [13], [14] our approach does not distinguish a training and an inference phase. Almost Rerere applies an online learning method whereby the CRRs are built, updated and applied greedily upon arrival of the conflicts. As visible from the example of Table 1, this causes a problem of *cold start*, whereby a conflict is not resolvable because there are no previous similar examples, and a problem similar to *underfitting*, whereby the available CRRs have been built from an insufficient number of examples and thus do not apply to unseen conflicts well. Table 13 shows another example. The cluster C1 contains two conflicts, which the developer has solved in a consistent manner by fusing the colliding versions.

Almost Rerere derives a CRR that simply searches for the string "`null, `" and removes it. Such a rule generates correct resolutions for several similar conflicts. But when a conflict with a similar yet slightly different structure enters the cluster, the result becomes inaccurate, as shown in Table 13. The automatic resolution omits one parameter of the method call. This happens because the CRR matches two regions of the string and removes both of them. The resulting resolution is similar to the developer's one (over 0.9), but semantically incorrect. When provided with more and diverse samples, the algorithm might try to incorporate in the search rule a pattern of what can be found before and after the matching area, making it less likely to create unwanted matches.

Due to the described online learning approach the evaluation of Almost Rerere described in Section 3 is conservative and provides a lower bound for the attainable accuracy, because in the early processing phases few repetitive conflicts are seen and thus many resolutions are rejected. Adopting a separated training step would improve the number of correctly resolved conflicts. The conflict clusters could be bootstrapped with the conflicts occurred in the initial phases of the project development. In this step, the developer would solve conflicts manually and no CRR would be applied. After the bootstrapping phase, the CRRs learned from a more significant amount of examples could be employed. This approach is similar e.g., to the one described in [6], in which the authors train their system with the conflicts collected in the first 8 weeks of development and then test it on the conflicts occurred in the next 4 weeks. Another complementary approach to mitigate the greedy behavior of Algorithm 1 is to allow cluster merging and splitting. This can be done by monitoring a quality metrics on the cluster structure (intra-cluster heterogeneity and inter-cluster similarity) so as to split a cluster when it exceeds an internal heterogeneity threshold and merge two or more clusters when they reach a sufficient similarity value. The determination of the optimal split and merge points could be formulated as a reinforcement learning problem. Both training and cluster reorganization are part of our future work described in Section 7.

5.3 External validity

The results obtained in the application of Almost Rerere on the 25 projects in the data set should not be extended blindly to other distributed development scenarios. 18 projects out of 25 involve the Java language and even if the accuracy of seven non-Java projects is in line with that of Java projects, an influence of the programming language(s) on accuracy cannot be excluded. Another factor to consider is that CRRs are learned in the context of a specific project and their generalization to different projects is probably difficult and surely deserves further investigation.

5.4 Reproducibility

The code of Almost Rerere and the data set used to produce the analysis tables reported in the paper are available at the following address:
https://github.com/herrera-sergio/AlmostRERERE
Almost Rerere will be published as an open source plug-in of the Git system.

6 Related Work

In this section we survey the research fields relevant to our work: the identification of code similarities and the classification of code samples according to a distance measure, the generation of mapping rules from examples of the input and of the desired output, the characterization of merge conflicts in distributed projects, the tools for supporting the merge process, and the data-driven methods for merge conflicts resolution.

6.1 Code similarity and clustering

Code similarity has been studied for software analysis, evaluation of refactoring issues, clone and plagiarism detection. *Textual approaches* use plain string matching for distance computation. Available metrics include Jaccard Coefficient [15], Levenshtein Distance [16], Longest Common Subsequence (LCS) [17], Jaro [18] and Jaro-Winkler [8], Needleman Wunsch [19] and Smith Waterman [20]. Ducasse et al. [21] used string-based Dynamic Pattern Matching (DPM) to detect code clones. Marcus & Maletic [22] applied latent semantic indexing (LSI) for finding similar code segments. *Lexical approaches* transform the code into sequences of tokens and compare the resulting vectors based on the duplicate sub-sequences. Lexical approaches are less sensitive

Cluster C1	CRR regex: "nul++, " replacement: ""
V1	`generateConstructor(typeNode, level, null, onConstructor, name);`
Res	`generateConstructor(typeNode, level, onConstructor, name);`
V1	`generateConstructor(typeNode, this.level, null, onConstructor, name);`
Res	`generateConstructor(typeNode, this.level, onConstructor, name);`
► A new conflict arrives	
V1	`generateConstructor(typeNode, null, null, onConstructor, name);`
V2	`generateConstructor(typeNode, level, onConstructor);`
When the CRR is applied to V1 it produces the following resolution	
Aut. res	`generateConstructor(typeNode, onConstructor, name);`
The automatic resolution is not accepted by the developer who commits a different one	
Man res	`generateConstructor(typeNode, null, onConstructor, name);`

Table 13: Example of conflicts generating a simple CRR and the incorrect resolution generated by Almost Rerere

to formatting and variable renaming. Example of lexical tools include CCFinder [23], DUP [24] and CP-Miner [25]. *Syntactic approaches* parse the code into an Abstract Syntax Tree (AST). ASTs are analysed with tree matching and comparison metrics. Syntactic approaches abstract variable names, literals and other code elements as tree nodes, allowing better detection of similarities when element names and values change. Several implementations exist. Koschke et al. [26] represent sub-trees as serialized token sequences to avoid the complexity of comparing sub-trees. Jiang et al. [27] encode sub-trees as feature vectors in an Euclidean space and apply locality-sensitive hashing to group similar vectors. Mayrand et al. [28] propose a distance measure calculated from names, layouts, expression and control flow structures. Kontogiannis et al. [29] use dynamic programming to compare begin-end blocks at statement level using the minimum edit distance. Davey et al. [30] train neural networks to find similar blocks of code based on automatically extracted features.

In this paper we need to determine code similarity in an online manner, by computing the distance between an incoming conflict and the code chunks of previously resolved conflicts. We use the textual approach for reasons of efficiency and apply the Jaro-Winkler distance metrics, which gave the best results experimentally.

6.2 Rule generation from examples

The problem of synthesizing string to string transformations from a set of input/output examples is NP-Complete [31]. Some tools have been developed to solve specific issues related to code editing with a Programming By Example approach. LAPIS [32] supports several semi-automatic string manipulations. One can either specify an initial search and replace expression for the system to improve or provide positive and negative examples of strings for inferring similar textual fragments. LASE [33] uses a syntactic approach to create a context-aware edit script from examples and exploits such script to identify edit locations and transform the code. The approach was later extended with RASE [34], an automatic refactoring tool for clone removal. It extracts common code by analysing systematic edits, creates new types and methods as needed, parameterizes differences in types and methods, and inserts return objects and exit labels based on the control and data flow. A more general approach is proposed in [9] and [10] in which genetic programming and cooperative co-evolution are used for synthesizing search and replacement patterns from examples of inputs and outputs. The search pattern is a regular expression that defines the portions of the string to be replaced and the portions to be reused by the replacement pattern. Programming by Example is exploited in [6] to create merge conflict resolution rules. We review this work in detail in Section 6.4.

In this paper we adapt the method of [9] and [10] to the case of CRR generation, by using the conflict as input and the resolution as output.

6.3 Conflict analysis

In recent years merge conflicts have been studied to understand the causes of their occurrence and find ways to prevent and resolve them. Dias et al. [35] investigate the factors that affect merge conflicts in 125 GitHub projects that adopt the MVC pattern. The probability of conflicts is found to increase with the lack of modularity and the size of the contribution (number of developers, of files and of changed lines) and with the time elapsed from the creation of a branch to its integration. Ghiotto et al. [5] manually reviewed five open source projects with great detail and then automated the analysis and extended it to 2.731 projects, with the objective of characterizing the merge conflicts and the developer's resolution strategies. They classified the merge conflicts along a number of dimensions: 1) the number of conflicting files per merge and the number of conflicting code chunks. 2) The involved language constructs, such as method declarations, variable declaration, if statements, etc. 3) The developer's resolution strategy chosen among five cases (described in Section 2). 4) The difficulty of the conflict, based on the resolution strategy adopted. The findings that motivate our work are that merge conflicts are frequent (25.328 failed merges and 175.805 conflicting chunks in 2.731 projects), have mostly a small size (median size is 2 or 2.5 LOCs), involve only a few constructs and are most frequently resolved by reusing code that exists in one of the two merged versions (in 87% of the cases). In [6] the authors analyse the merge history of the Microsoft Edge browser and find results that confirm the observations of [5]: $\approx 28\%$ of the conflicts involve 1 or 2 lines and many resolutions follow the same pattern. An alternative view on the conflict management process is provided in [36] which reports the findings of an empirical study conducted through interviews and questionnaires with developers. The study highlights that conflict management follows a cycle of development, awareness, planning, resolution and evaluation. It reveals a rather limited propensity to proac-

tively monitor merge conflicts during development and a tendency to defer resolution, which impacts the workflow of the entire team. A reported difficulty that motivates deferral is the complexity of understanding the collisions between versions, which motivates the utility of tools capable of learning how collisions have been resolved in the past and of supporting developers in their resolution planning task. The planning phase is the focus of the work in [37], which applies supervised machine learning algorithms to predict the difficulty level of a conflict from the statistical data about the merge conflicts collected from 128 Java projects. The best classifier, based on bagging, achieves 79% precision at 79% recall and exploits 10 attributes to characterize the conflicts. Knowing the complexity of a conflict can help developers plan the resolution time and build the team responsible of addressing the conflict.

Our work is motivated by the findings of [5] and [6] and confirms the relevance of small size merge conflicts with a repetitive resolution pattern.

6.4 Merge support and conflict prevention tools

Conflict analysis and visualization tools assist the developer in the merging process to reduce the number of conflicts to be resolved manually [38].

Unstructured tools apply textual approaches to identify the change-set and help developers merge branches. Most tools use 3-way merge based on the Diff3 algorithm [39]. This technique is used by the *git merge* command[8] and incorporated in several IDEs. Unstructured approaches are fast and language independent, but they can not handle all types of conflicts, such as those that emerge from refactoring [12]. Structured approaches exploit the syntax and semantics of the artifacts. They represent the versions as trees or graphs and merge them using tree matching algorithms and language information. JDime [40] is a tree matching merging tool that proved able to decrease by 40% the number of conflicts w.r.t. to unstructured merge techniques and to reduce the number of lines in the conflicts. The approach of [41] exploits version space algebra. When a conflict is detected that cannot be solved by the structured merger, the algorithm creates a space representation of all the possible resolutions and ranks them based on the probability of its elements to appear in the solution. This technique resolved 95% of the conflicts in the test data set but assumes that the resolution is a combination of the versions and cannot scale to complex conflicts which make the solution space intractable. Semi-structured approaches combine structured and unstructured techniques; they partially analyse the syntactic structure and semantics of the artifacts and apply textual methods to the parts not addressable by the semantic analysis. The FSTMERGE [42] tool transforms the source files into program structure trees in which the internal nodes represent program elements (classes and methods) and the leaves represent the content of the methods as plain text. Structured merge is applied to the nodes and unstructured merge to the leaves. In the evaluation [43] this approach reduced the number of conflicts w.r.t. unstructured merge by 34% and the number of conflict lines by 28%. Cavalcanti et al. [44] proposed jFSTMERGE for mitigating false positives (conflicts that can be solved simply by ordering the lines) and false negatives (code that can be merged but produces an invalid program) by means of a handler process associated with each type of conflict that applies textual analyses and compiler functions. This improvement reduces the number of conflicts w.r.t. the original implementation by 36%. IntelliMerge [12] focuses its graph-based approach on the detection of conflicts caused by code refactoring; first the three input artefacts are transformed into a graph representation; then the best matching for vertices is searched and the unmatched vertices are classified by type context and body similarity to identify refactoring problems and obtain the best match. Next a new graph is created by combining the matched vertices and identifying the possibly remaining conflicts. Finally the resulting graphs is transformed into text. The evaluation showed a reduction in the number of conflicts by 58.9% w.r.t. unstructured tools and 11.84% w.r.t. jFSTMerge. An alternative approach is to prevent merge conflicts by enabling the semi-synchronous collaboration of concurrent developers. The pioneering work [45] illustrates a collaboration model in which developers working asynchronously on a shared code base are alerted of potential conflicts before merge time and can activate synchronous collaboration sessions to anticipate the collision and reduce the insurgence of conflicts. The work in [46] takes a different approach to conflict mitigation. Early alerts about conflicts are not exploited to trigger synchronous reconciliation sessions but are exploited together with project dependency information to derive task constraints so that distributed developers can organize their work according to a conflict-free schedule.

Our approach has a different yet correlated goal: rather than reducing the occurrence, frequency and size of conflicts, Almost Rerere aims at learning the way in which developers resolve them.

6.5 Conflict resolution synthesis

Recently the problem of learning how to solve merge conflicts from the code provided by the developers has been formulated and addressed independently in [6] and in [47]. In [6], Pan et al. propose a program synthesis approach for learning the resolution of conflicts located in the include and macro sections of C++ programs. The method requires a Domain Specific Language (DSL) to express the patterns to search in the conflict chunks and the transformation operations for generating the resolution (copy, concat and move); then a set of input-output examples are given as input to the PROSE[9] inductive synthesizer which exploits a top-down approach that learns efficient inverse functions for the DSL operators and generates the transformation program mapping each input conflict into the corresponding resolution. The input examples are derived from the Microsoft Edge repository. The evaluation reported that the synthesized program could resolve 11.44% of the conflicts. The accuracy for the specific class of conflicts addressed is 93.2% and the approach performs better with 1-2 lines conflicts. The approach of [6] and our method are complementary: the former uses language patterns and a program synthesis

8. https://git-scm.com/docs/git-merge

9. https://github.com/Microsoft/prose

technique, whereas our work is language independent and uses regex synthesis. A hybrid perspective between conflict identification and resolution is explored in the data driven DEEPMERGE tool [47], which applies a Deep Learning approach to identify and solve spurious conflicts, i.e., conflicts that can be resolved by line reordering and concatenation. The novelty consists of encoding the conflicts into an edit-aware matrix and use such representation to train a Deep Learning model that predicts which lines of the input shall appear in the output. The evaluation exploits a data set of JavaScript programs to assess the effectiveness of DeepMerge in identifying conflicts requiring manual resolution. In this task the authors report a precision of 78% and a recall of 73%. The approach can also be used to predict a conflict resolution, limited to the case in which the output lines are a combination of the input lines, possibly with one inserted token. In this specific case, the prediction scores 72% precision at 34% recall.

6.6 Automatic bug fixing

A closely related problem is that of learning how to correct bugs based on the history of previous error fixes. Automatic conflict resolution is more comprehensive because the colliding updates can represent either a bug fix or a more general update, e.g., the change of an imported library. The Getafix system [14] features an approach for producing human-like bug fixes by learning from a repository of previous error corrections. Getafix and Almost Rerere share the same goal of learning useful code updates from past examples but differ in several aspects: 1) Getafix exploits as input a buggy piece of code, the fix and the information of the error type. Almost Rerere exploits as input the two colliding pieces of code and the manual resolution. The collision may be due to a bug fix or to a more general code update. 2) Both Getafix and Almost Rerere give as output the modification to apply to the code to address the problem (bug fix or conflict reconciliation). 3) Getafix represents the code by means of an Abstract Syntax Trees (AST), Almost Rerere as plain text. 4) Getafix represents the modification as a sequence of AST edit steps, Almost Rerere as a search and replacement regular expression. 5) Getafix distinguishes a learning phase and an inference phase. Almost Rerere uses a continuous online learning approach without a pre-training phase. 6) Getafix generalizes concrete cases into patterns by means of anti-unification. Almost Rerere exploits genetic programming to produce CRRs applicable to similar conflicts. 7) Getafix uses hierarchical clustering to compute multiple generalizations at different abstraction levels from the same type of bug. Almost Rerere uses (non-hierarchical) one-pass online clustering to group similar conflicts. The cluster can be seen as the counterpart of a Getafix pattern, but the Almost Rerere cluster is learned solely from data, and thus multiple non related clusters could be produced for the same type of bug. 8) Getafix has multiple rules for the same type of bug and thus can create and rank multiple resolutions. Almost Rerere assigns a conflict to the most similar cluster and thus computes the top-1 recommendation. To provide the top-k resolutions Almost Rerere could apply the CRRs of the most similar k clusters. 9) Getafix validates the proposal w.r.t. to the input error type via static analysis. Almost Rerere does not know the error type and performs no validation. 10) Getafix achieves 30% accuracy over 6 categories of bugs in 1,268 samples. Almost Rerere achieves **54,86%** accuracy over a not a priori fixed set of error types from 14,872 conflicts. The authors of [13] apply Neural Machine Translation (NMT) to map buggy code into corrected code. Their work has points in common and significant differences with respect to Almost Rerere: 1) both systems take as input pairs of code snippets <before state, after-state>. 2) [13] represents the inputs as a sequence of AST tokens with recurring identifiers and literals abstracted as idioms, whereas Almost Rerere learns directly from the code. 3) [13] distinguishes a training and an inference phase, whereas Almost Rerere applies continuous learning. 4) [13] computes the fixes with a black-box neural encoder-decoder architecture, whereas Almost Rerere adopts a white box approach in which the mapping of the before state into the after state is computed by regex, which are human-readable and even customizable by developers if needed. 5) [13] uses beam search in the neural architecture to output multiple fixes, whereas Almost Rerere computes only the top-1 reconciliation. 6) When looking at the top-1 and top-5 fixes the accuracy of [13] ranges from 9,22% to 27% on small snippets, whereas Almost Rerere has a top-1 accuracy of 62.29% on SL conflicts and of 46.50% on ML conflicts.

6.7 Contribution

In this work we have addressed the problem of reconciling merge conflicts by exploiting the knowledge embedded in past developers' decisions. The proposed technique is purely data-driven and employs a learning process that generates conflict resolution rules from examples of conflict chunks and of manual solutions. It does not require the a priori identification of patterns as in [6] but encodes the repetitive nature of conflicts into conflict clusters built using a similarity measure. It does not impose restrictions on the type of resolution that can be actuated by the rules, as in [6] and [47], and on the program constructs that may cause the conflict, as in [6]. It is text-based and language independent and can apply to artifacts that mix different syntax, such as scripted HTML templates. It can be applied in a purely online mode, as demonstrated in the evaluation, by letting the system build clusters as the conflicts arrive. Or it can be employed in a partially offline mode, similar to the training phase of [6] and [47], by allowing a bootstrap step in which the system silently observes the manual resolutions and builds an initial more representative set of conflict clusters.

In our previous work [48] we applied a preliminary version of Almost Rerere to the conflicts extracted from an initial data set of 5 Java projects and reported a first qualitative evaluation of the potential of the data-driven synthesis of conflict resolution rules. In this work we extend [48] as follows

- We have improved the similarity search, by comparing also the after state (resolution) of a conflict with those present in the clusters, and fine-tuned the boost factor of the Jaro-Winkler distance via parameter search.
- We have significantly expanded the evaluation (25 instead of 6 projects, in 4 languages instead of 1)

and reported the overall accuracy (percentage of identically resolved conflicts).
- We have assessed the impact of such dimensions as the conflict size, the resolution strategy and the language constructs on the percentage of resolved conflicts and on the accuracy.
- We have provided a complete example of how the conflict resolution process works.
- We have exposed the role of the domain knowledge and of the learning method in the conflict resolution process and discussed the threats to validity of our work in depth.
- We have published the code and the conflict data set at the base of our work, for fostering reproducibility and the comparison of other methods to our text-based approach.

7 Conclusions and future work

The paper describes an approach for the automatic resolution of merge conflicts during code integration. The approach is based on the synthesis of a search and replacement expression from previously resolved similar conflicts. The approach has been evaluated on 14.872 small size (up to 6 LOCs) conflict chunks extracted from 25 projects (18 Java, 3 JavaScript, 2 PHP, 2 Python). The results show that the system can handle $\approx 49\%$ of the conflicts produced during the merge process ($\approx 88\%$ if one considers conflicts that have at least one similar conflict in the data set) and can reproduce exactly the same solution of the human developers in $\approx 55\%$ of the cases ($\approx 62\%$ for single line conflicts). A preliminary analysis of non-identical resolutions suggests that accuracy may underestimate the utility of the synthesized resolutions, because there are cases in which the automatic resolution is equivalent to the manual one, albeit text-wise different. The proposed approach is implemented in an open source Git utility, called *Almost Rerere*, which extends the *Git Rerere* functionality.

Future work will focus on the following directions:

- Assessing the impact on accuracy of a hybrid execution strategy that mitigates the purely online approach used in the illustrated evaluation. The present version of the CRR generation algorithm assigns conflicts to the currently most similar cluster and never reconsiders such an allocation. This may lead to the progressive loss of internal coherence of some clusters, which in turn may affect the accuracy of the generated CRRs. We plan to experiment two distinct approaches. On one side, we will evaluate the benefit of an initial training step during which the system receives conflicts in a batch and builds an initial set of clusters without attempting to resolve conflicts. In this way, the early conflicts, which have no antecedents and thus determine a loss of accuracy, will not penalize the system but will be exploited to create a robust set of examples. On the other side, we will assess the benefits of monitoring the evolution of intra-cluster heterogeneity and inter-cluster similarity to make decisions about splitting or merging clusters. This approach will be formalized as a reinforcement learning problem so as to identify an optimal strategy for updating the cluster structure.
- The investigation of the portability of CRRs across projects. The main question is whether there exist resolution patterns that apply beyond the boundaries of a single project and how to recognize them and transfer knowledge from one project to another.
- The enrichment of the CRR language with a small set of general purpose features (e.g., integer arithmetic), which could help the developer edit the generated CRR and extend its applicability.
- A large scale evaluation of the approach on a massive amount of projects and conflicts and the publication of the CRRs that proved most valuable within specific projects and across projects.

REFERENCES

[1] C. R. De Souza, D. Redmiles, and P. Dourish, "Breaking the code, moving between private and public work in collaborative software development," in *Proceedings of the 2003 International ACM SIGGROUP conference on Supporting group work*. ACM, 2003, pp. 105–114.

[2] W. F. Tichy, "Rcs—a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[3] H. Le Nguyen and C.-L. Ignat, "An analysis of merge conflicts and resolutions in git-based open source projects," *Computer Supported Cooperative Work (CSCW)*, vol. 27, no. 3-6, pp. 741–765, 2018.

[4] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 180–190.

[5] G. Ghiotto, L. Murta, M. Barros, and A. Van Der Hoek, "On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 892–915, 2018.

[6] R. Pan, V. Le, N. Nagappan, S. Gulwani, S. Lahiri, and M. Kaufman, "Can program synthesis be used to learn merge conflict resolutions? an empirical analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 785–796.

[7] C. C. Aggarwal and K. Subbian, "Event detection in social streams," in *Proceedings of the Twelfth SIAM International Conference on Data Mining, Anaheim, California, USA, April 26-28, 2012*. SIAM / Omnipress, 2012, pp. 624–635. [Online]. Available: https://doi.org/10.1137/1.9781611972825.54

[8] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage." *Proceedings of the Section on Survey Research Methods*, 1990.

[9] A. Bartoli, A. D. Lorenzo, E. Medvet, and F. Tarlao, "Inference of regular expressions for text extraction from examples," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1217–1230, May 2016.

[10] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Automatic search-and-replace from examples with coevolutionary genetic programming," *IEEE transactions on cybernetics*, 2019.

[11] K. Dreßler and A.-C. N. Ngomo, "Time-efficient execution of bounded jaro-winkler distances," in *OM*, 2014.

[12] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "Intellimerge: A refactoring-aware software merging technique," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[13] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019. [Online]. Available: https://doi.org/10.1145/3340544

[14] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 159:1–159:27, 2019. [Online]. Available: https://doi.org/10.1145/3360585

[15] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 2008, pp. 257–266.

[16] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[17] N. Nakatsu, Y. Kambayashi, and S. Yajima, "A longest common subsequence algorithm suitable for similar text strings," *Acta Informatica*, vol. 18, no. 2, pp. 171–179, 1982.

[18] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida," *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989.

[19] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[20] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[21] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 1999, pp. 109–118.

[22] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE, 2001, pp. 107–114.

[23] 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*, vol. 28, no. 7, pp. 654–670, 2002.

[24] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE, 1995, pp. 86–95.

[25] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[26] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 253–262.

[27] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[28] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics." in *icsm*, vol. 96, 1996, p. 244.

[29] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1-2, pp. 77–108, 1996.

[30] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley, "The development of a software clone detector," *International Journal of Applied Software Technology*, 1995.

[31] J. Hamza and V. Kunčak, "Minimal synthesis of string to string functions from examples," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2019, pp. 48–69.

[32] R. C. Miller and B. A. Myers, "Lapis: Smart editing with text structure," in *CHI Extended Abstracts*, 2002, pp. 496–497.

[33] N. Meng, M. Kim, and K. S. McKinley, "Lase: locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 502–511.

[34] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 392–402.

[35] K. Dias, P. Borba, and M. Barreto, "Understanding predictive factors for merge conflicts," *Information and Software Technology*, vol. 121, p. 106256, 2020.

[36] N. Nelson, C. Brindescu, S. McKee, A. Sarma, and D. Dig, "The life-cycle of merge conflicts: processes, barriers, and strategies," *Empir. Softw. Eng.*, vol. 24, no. 5, pp. 2863–2906, 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9674-x

[37] C. Brindescu, I. Ahmed, R. Leano, and A. Sarma, "Planning for untangling: predicting the difficulty of merge conflicts," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 801–811. [Online]. Available: https://doi.org/10.1145/3377811.3380344

[38] K. N. A. Adam and N. Károly, "Merging problems in modern version control systems," *Multidiszciplináris Tudományok*, vol. 10, no. 3, pp. 365–376, 2020.

[39] S. Khanna, K. Kunal, and B. C. Pierce, "A formal investigation of diff3," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 2007, pp. 485–496.

[40] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, 2015.

[41] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[42] S. Apel, J. Liebig, C. Lengauer, C. Kästner, and W. R. Cook, "Semistructured merge in revision control systems." in *VaMoS*, 2010, pp. 13–19.

[43] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: rethinking merge in revision control sys-

tems," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 190–200.
[44] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–27, 2017.
[45] P. Dewan and R. Hegde, "Semi-synchronous conflict detection and resolution in asynchronous software development," in *Proceedings of the Tenth European Conference on Computer Supported Cooperative Work, 24-28 September 2007, Limerick, Ireland*, R. Harper and C. Gutwin, Eds. Springer, 2007, pp. 159–178. [Online]. Available: https://doi.org/10.1007/978-1-84800-031-5_9
[46] B. K. Kasi and A. Sarma, "Cassandra: proactive conflict minimization through optimized task scheduling," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 732–741. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606619
[47] E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. K. Lahiri, "Deepmerge: Learning to merge programs," *arXiv preprint arXiv:2105.07569*, 2021.
[48] P. Fraternali, S. L. H. Gonzalez, and M. M. Tariq, "Almost rerere: An approach for automating conflict resolution from similar resolved conflicts," in *Web Engineering - 20th International Conference, ICWE 2020, Helsinki, Finland, June 9-12, 2020, Proceedings*, ser. Lecture Notes in Computer Science, M. Bieliková, T. Mikkonen, and C. Pautasso, Eds., vol. 12128. Springer, 2020, pp. 228–243. [Online]. Available: https://doi.org/10.1007/978-3-030-50578-3_16

Piero Fraternali Full professor of Web Technologies at Politecnico di Milano. Author of more than 300 papers in international peer reviewed conferences and journals, and a number of books. His main research interests concern methodologies and tools for WEB/mobile application development, socio-technical system design, gamification and serious games. He is co-author of OMG's IFML standard (http://www.omg.org/spec/IFML/) and co-founder of WebRatio (http://www.webratio.com), a start-up focused on the commercialization of a tool suite for the Model-Driven Development of Web/mobile cloud-powered applications.

Sergio Luis Herrera Gonzalez PhD and research assistant in the Web data and society group at Politecnico di Milano. He is working in the development of novel methodologies and code generation tools for the semi/automatic development of web and mobile applications with a special focus on user/centric applications with gamified functions.