Almost Rerere: an approach for automating conflict resolution from similar resolved conflicts

Piero Fraternali, Sergio Luis Herrera Gonzalez and Mohammad Manan Tariq

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano,
Piazza Leonardo da Vinci 32, Milan, 20133,Italy.
piero.fraternali@polimi.it, sergioluis.herrera@polimi.it,
mohammad.tariq@mail.polimi.it

Abstract. Concurrent development requires the ability of reconciling conflicting updates to the code made independently. A specific case occurs when long living feature branches are integrated to a rapid changing code base. In this scenario, every integration test will require to manually resolve the same conflicts at every iteration. In this paper we propose a framework for automating the detection and resolution of conflicts in the code updated by distinct developers, one of which may be a code generator. The tool learns how to solve conflicts from past experience and applies resolutions, encoded as replacement regular expressions, to conflicts not seen before. Experiments show that the number of automatically resolved conflicts and the quality of the solution increase as the system acquires experience.

Keywords: Automatic Conflict Resolution · GIT · Code integration.

1 Introduction

The development of large and complex software applications requires distributing programming tasks among multiple developers. In Model Driven Development, this scenario may also include code generators that produce implementation code from high level models. When the same code base is updated concurrently by different actors, whether human or automatic, the possibility arises that the same portion of the code is affected, generating inconsistencies between the changes made by the actors and/or the code base. This occurrence is called *conflict* [8].

Conflict management is particularly relevant in the engineering of Web and multi-channel applications, because the implementation of the functional and of the presentation requirements is often assigned to distinct developers working on the same code base. Albeit the presentation aspects of Web-based interfaces can be factored out in CSS rules, the separation of concern is in reality partial, because it is a common practice to add presentation-oriented elements to the page structure to support the selective application of presentation styles.

Therefore, the concurrent update of structure and of the presentation aspects produces conflicts, which require the continuous alignment of the two facets of development to preserve the change of either aspect.

To support distributed development, Version Control Systems (VCS) [29] offer functions to share code, track changes, and identify conflicts. When conflicts are signalled by the VCS, the resolution is delegated to the developer, which makes code integration a time-consuming task [17]. Conflict resolution is also repetitive because similar or identical conflicts appear at every iteration. A significant case occurs in Model Driven Development when the source model and/or the model-to-text transformation templates are modified. In this case, the code generator applies the same transformation rules to many spots in the code overwriting the manually integrated code and producing multiple changes with the same pattern. In our previous work [6], we have addressed the management of the conflicts between handwritten and generated code, albeit the Virtual Developer approach helps reducing the conflicts between handwritten and generated code, still the need persists of manually resolving many similar conflicts.

In this paper we develop a method to let a VCS learn how to resolve similar conflicts. A conflict and its resolution can be modelled as a pair (*before-state*, *after-state*), where the before state contains the line(s) of the code affected by concurrent inconsistent updates and the after state comprises the code provided by the developer to resolve the inconsistency. The key idea is to exploit the conflict resolutions implemented by human developers in the past to create rules applicable to future (similar) conflicts. Intuitively, this requires the following process. When the first conflict is resolved manually, its resolution pair is processed to derive a *Conflict Resolution Rule (CRR)*. Then the first *Conflict Cluster (CC)* is created and the rule is associated with it. When a new resolved conflict arrives, its before state is compared with the existing CCs. If it is similar to some existing CC, it is added to it and the CRR associated with the CC is applied to resolve it; otherwise, the user is prompted to provide a resolution and a new (CC, CRR) pair is created. A quality metrics on the resolution provided by the CRR is monitored; as the system observes more and more manual conflict resolutions, the quality of the resolution computed by the CRR increase and the user may accept that the rule is applied without supervision.

The contribution of the paper can be summarized as follows: 1) We introduce the problem of automating the resolution of similar conflicts in concurrent application development and define the version control framework and workflow needed to handle it. 2) We apply the Hierarchical Agglomerative Clustering (HAC) algorithm with the Jaro-Winkler string similarity measure [30] to group similar conflicts in Conflict Clusters. A CC includes conflicts that may be resolved by the same rule. 3) We adapt the approach of [2] and [3] to automatically synthesize Conflict Resolution Rules for the conflicts of a CC. A CRR is a search

and replace regular expression extracted from a set of conflict resolutions specified as pairs (*before-state*, *after-state*). Specifically, the CRR is the best fitted search and replace expression that maps the before states of all the conflicts in the CC into the respective after state. 4) 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. 5) We evaluate the approach in the development of a web application using a Model-Driven Development tool that generates conflicts with the handwritten code and by extracting conflicts and resolutions from the history of submissions of large Git open source project repositories.

2 Related Work

The relevant related work refers to the identification of code similarities and to the generation of string rewriting rules from input/output examples.

Code similarity has been studied for software analysis, evaluation of refactoring issues, clone and plagiarism detection, etc. Textual approaches use direct string matching and comparison techniques for the detection of similarities, a wide set of this type of algorithms is available e.g. Jaccard Coefficient [19], Levenshtein Distance [18], Longest Common Subsequence (LCS) [26], Jaro [11] and Jaro-Winkler [30] similarity, Needleman Wunsch algorithm [27], Smith Waterman algorithm [28], etc. Ducasse et al. [9] used string-based Dynamic Pattern Matching (DPM) to detect code clones. Marcus & Maletic [21] applied latent semantic indexing (LSI) for finding similar code segments. Token-based approaches use lexical analysis for transforming the code into sequences of lexical tokens and the resulting sequences are then compared searching for duplicated subsequences of tokens. Tools implementing this approach are CCFinder [14], DUP [1] and CP-Miner [20]. Syntactic approaches use parsing to convert the source code into an Abstract Syntax Tree (AST). ASTs can then be analysed using tree-matching [4,15,12] and metrics-based methods [22]. The above-described approaches are combined with advanced clustering techniques, e.g., to provide intelligent recommendations or to identify bugs automatically. Kreutzer et al. created C3 (Clustering of Code Changes) a tool that scans code repositories to

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

automatically detect code fixes by clustering code changes using diff-based and AST-based metrics.

The problem of synthesizing string-to-string transformations from a set of input/output examples is NP-Complete [10]. Nevertheless, some approaches have been developed to solve specific instances of the problem, many of them related to code editing. LAPIS [25] uses an assisted approach in which the user provides an initial search & replace expression that the system can improve or a set of positive and negative examples that are used to infer similar sections of the text. LASE [24] uses a syntactic approach to create a context-aware edit script from examples and uses the script to automatically identify edit locations and apply the transformation to the code. The approach was later extended with RASE [23], an automatic refactoring tool for clone removal. A different approach was proposed by Bartoli et al. in [2] and [3], in which they used Genetic Programming (GP) and cooperative co-evolution to synthesize search & replacement patterns based on examples of the desired behaviour. The search pattern is a regular expression (regex) that defines the portions of the string to be replaced and the portions to be reused by the replacement pattern.

In this work, we aim at developing a language-independent tool able to handle conflicts in modern web and mobile projects mixing several languages. We base our tool on textual approaches for the similarity computation and conflict clustering tasks and adapt the algorithm of Bartoli et al. for conflict resolution. A text-based method presents advantages over syntactic techniques that require the creation of AST and are language-dependent. It can be applied to any semi-structured string, is independent of the text format, and can perform context-dependent rule extraction.

3 Background

3.1 Conflict Resolution

The concurrent development of software applications requires the management of possibly conflicting updates to the same code base by different developers. A typical workflow, in which the same code base is updated inconsistently by two developers (D1 and D2) producing a conflict, proceeds as follows.

Developers D1 and D2 initialize their local code base C1 and C2 from the current content of the central code-base C, which comprises the status of the project resulting from n preceding revisions. D1 and D2 start working independently on their local revisions, R^1_{D1} and R^1_{D2}, initially equal. D1 introduces a new feature by applying changes to R^1_{D1}, creating a new local revision R^2_{D1}. Next, D2 independently updates R^1_{D2} to introduce another feature, creating a new local revision R^2_{D2}. D1 submits his local revision to the central code-base generating a new shared revision R^2_{C}. No conflicts arise because D1 applied his update to the

shared consolidated revision ($R^1_C = R^1_{D1}$) . Now D2 submits his revision to the central code-base. The operation creates a conflict because the submitted revision does not derive from the *current* shared revision ($R^2_C \neq R^1_{D2}$). D2 performs conflict resolution and generates a new local revision R^3_{D2}, which integrates the feature locally developed by D2 and the current state of the central code-base (which comprises the feature developed and submitted by D1). He submits R^3_{D2} to the central code-base and produces a new shared revision R^3_C.

The following example illustrates the content of a conflict. The markers <<<<<<< and >>>>>>> delimit the conflict area, and inside the conflict area the marker ======= separates the two colliding updates.

```
<h3>List</h3>
<table class="table table-hover table-condensed">
<!-- Conflict area Start -->
<<<<<<<
    <thead class = "header">
=======
    <thead class= "table_header">
>>>>>>>
<!-- Conflict area End -->
        <tr>
            <th>#</th>
            <th>First Name</th>
            <th>Last Name</th>
        </tr>
    </thead>
    ...
</table>
```

The conflict illustrated above can be resolved by the following CRR:

```
{
    "regex": "(?:([^2])[^_]eader(\"))++",
    "replacement": "$1table_header$2"
}
```

The above CRR searches for the character sequence “eader” preceded by a case insensitive character and between quotes, and replaces the characters between the quotes with the string "table_header", leaving the rest unmodified.

3.2 Git Rerere

Git Rerere (**RE**use **RE**corded **RE**solution)[2] is a component of Git 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 the same conflict occurs again, Git Rerere reuses the recorded solution to manage the conflict automatically. Git Rerere is typically used in the development a long-lived feature branch where developers execute several testing cycles before the release of the feature. The pre- and post-images of a conflict are stored in a sub-directory named by *hashing* the content of the conflict area of the file. When a conflict occurs, Rerere extracts the conflict area, generates the hash, searches for the directory with that name, extracts the post-image and uses it to perform a merge with the current state of the file, thus resolving the conflict and preserving the rest of the non-conflicting changes. Git Rerere is designed to automate the resolution of multiple identical conflicts and cannot handle similar pre-images to apply a recorded solution to a non-identical conflict. This aspect shows up also in the internal organization of the tool. Due to the use of conflict hashes as access keys to the directory organization, any change in the hash of a conflict prevents finding the recorded solution. Moreover, when multiple conflicts in a source file occur, Rerere generates a single hash per source file using all conflict areas. If a new conflict occurs in a file where a previously resolved conflict existed, a new hash is created and the previously resolved conflict is no longer retrieved.

Almost Rerere aims at resolving automatically not only the conflicts that are *identical* to previously seen instances, but also those that are *similar* to instances solved in the past. It identifies conflicts with the same pattern, clusters them based on a similarity criterion, and associates each cluster with a rule synthesized from the conflict resolutions of the cluster. It does not depend on the stability of the *hash* of the conflicts but rather exploits the changes in the conflict text to learn a pattern that characterizes a family of related conflicts and to build a replacement rule that can be applied to resolve future similar occurrences.

4 Proposed Approach

The proposed approach consists of three main steps: the identification of similar conflicts using the Jaro-Winkler string similarity metrics; 2) the grouping of similar conflicts using an agglomerative hierarchical clustering algorithm; 3) the synthesis of conflict resolution rules by giving clusters in input to a genetic algorithm that computes a search and replacement expression.

[2] https://git-scm.com/docs/git-rerere

4.1 Almost Rerere architecture

Figure 1 shows the architecture of Almost Rerere, which comprises four main components: the Submission Manager, the Cluster Manager, the CRR Generator, and the Conflict Resolver.

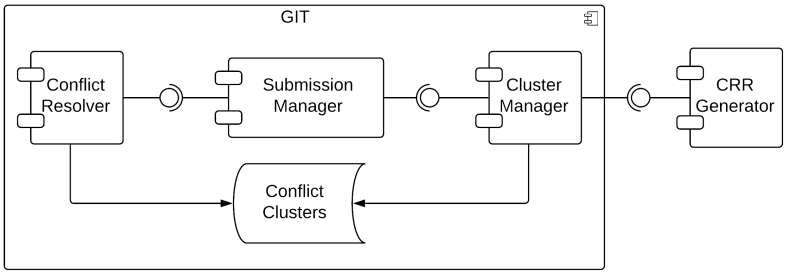

Fig. 1: *Almost Rerere* architecture

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 proposed in [3] and is triggered every time a conflict is added to a cluster. It synthesizes a CRR in the form of a regex & replacement expression that can be applied to the resolution of the conflicts of that 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.

4.2 Conflict Cluster Generator

The core contribution of Almost Rerere is the recognition that a new conflict is similar to an occurrence already addressed in the past, so that a generalization of the previously applied resolution can be reused to cope with the new conflict. Generalizing a CRR requires identifying a pattern common to multiple conflicts, which consists of a constant and of a variable part. The constant part is used to match the conflicts that can be addressed by the CRR. The variable part enables addressing the differences in the conflicts with the same pattern. This approach requires two elements: a metrics for quantifying the distance between conflicts and an algorithm to group conflicts based on such a distance.

Almost Rerere computes the distance between conflicts based on a string similarity measure. Several string similarity algorithm were evaluated on a test data set of about 200 code line pairs. The *Jaro-Winkler* similarity algorithm

showed the highest similarity scores of code lines in the same pair in 80% of cases and was selected for the implementation of the clustering algorithm. A similarity threshold was determine by calculating the precision, recall and F_1 for several thresholds, it was determine that 0.80 was the value maximizing F_1. The intuition behind the Jaro-Winkler algorithm performing better on the conflict data set 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 reserved words, e.g. type declarations (*int, double, String*), access declarations (*public, private, protected*), flow control specifications (*if, while, switch*), etc. that 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 more likely to be updated.

The conflicts are grouped using hierarchical agglomerative clustering (HAC) [13]. When a conflict with its respective resolution is received from the Submission Manager, the cluster with the highest similarity score is searched. If the similarity score of the retrieved cluster is below a threshold (0.80), a new cluster is generated and the conflict is assigned to it; otherwise the conflict is added to the cluster. In both cases, the CRR generator is called to create a new rule or an improved version of an existing rule for the cluster. Each cluster has a unique id and contains an array of objects composed by the conflict and its resolution. Figure 2 shows an example.

```
{
    "1": [
        {
          "conflict":"if (g instanceof UndirectedGraph){",
          "resolution":"if (g instanceof UndirectedGraph<?,?>){"
        },
        {
          "conflict":"if (!(graph instanceof DirectedGraph)){",
          "resolution":"if (!(graph instanceof DirectedGraph<?, ?>)){"
        },
        {
          "conflict":"if (this.graph instanceof UndirectedGraph){",
          "resolution":"if (this.graph instanceof UndirectedGraph<?,?>){"
        }
    ]
}
```

Fig. 2: Example of Conflict Cluster

4.3 CRR generator

The CRR Generator exploits the general-purpose string search & replacement algorithm of [3], which takes as input a series of examples, consisting of 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 that describes both the portions of the string to be replaced and those to be reused; the latter describes how to build the output string.

The method of [3] 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, validation and testing. 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.

The method of [3] 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. As an example of the generated CRR, Figure 3 shows the rule generated from the cluster of Figure 2.

```
{
    "1": [
         {
            "regex": "(h)(?=\))",
            "replacement": "$1<\?,\?>"
        }
    ]
}
```

Fig. 3: The CRR generated from the CC of Figure 2

The CRR searches for a character `h` followed by a closed parenthesis and the replacement expression then inserts the expression `<?,?>` after the character `h` to implement the desired transformation.

When the number of available samples is small, the algorithm is sensitive to the way in which the samples are assigned to the training, testing, and validation sets. To mitigate this problem, the samples are randomly divided into the training, testing and validation sets and the algorithm is executed multiple times. If the generated CRR is the same across the executions, which indicates

that the algorithm has converged, it is saved. Otherwise, all solutions are kept, and the CRR is composed as the disjunction of the computed expressions. In the experiments, two rounds of execution proved to afford the best trade-off between performance and accuracy of the synthesised CRRs.

4.4 Conflict resolver

This component has the responsibility to resolve the new conflicts when the developer executes a *git merge* command. The component searches for the cluster with the highest similarity measure with respect to the incoming conflict. The CRR of the selected cluster is applied and the result is returned to the developer.

5 Evaluation

Almost Rerere was evaluated in two case studies: the development from scratch of a web-based crowd sourcing platform using an Agile Model-Driven Development tool and approach, and the resolution of conflicts extracted from the reproduction of commits in the Git repositories of long-run open-source projects.

5.1 Integration of handwritten and generated code

The goal of the test was to make an evaluation of how Almost Rerere could help in resolving conflicts during the life-cycle of a Model-Driven Development project. The application was developed using IFMLEdit.org[3] [5], an online tool for the rapid prototyping of web and mobile applications based on the Interaction Flow Modeling Language (IFML)[7]. The developed application was a web-based crowd-sourcing platform for the selection and annotation of images. The development process of the application was divided into seven sprints. At each sprint, the developers applied changes to the IFML model, to the code generation templates, which combine HTML, JavaScript and CSS, and manually modified the automatically generated code to add non-modelled features. Two developers worked in parallel, the main repository of the code was the master branch, each developer worked on his own branch and integrated the changes to the master branch once completed. During the sprints, both developers updated the code generation templates. When the code was generated from the modified templates, the changes would propagate to all the relevant pages. In other cases, the changes were made directly on the generated code. Both developers committed changes to their local branch. When the local branches were merged into the main repository, conflicts arose because independent changes were applied to the same lines of code. Whenever a conflict was detected, Almost Rerere intervened

[3] https://ifmledit.org/

to resolve the conflict or to record the manual resolution provided by the developer. During the seven sprints about 200 conflicts were resolved. Figure 4 shows the total number of conflicts and the number of those resolved by Almost Rerere at each sprint. In the first sprint, it can be observed that no conflict is resolved, because no recorded conflicts existed at that point. In the second sprint, only 4 conflicts were resolved, because the number of samples available was small. As the number of occurring conflicts increased, also the number of resolutions by Almost Rerere grew.

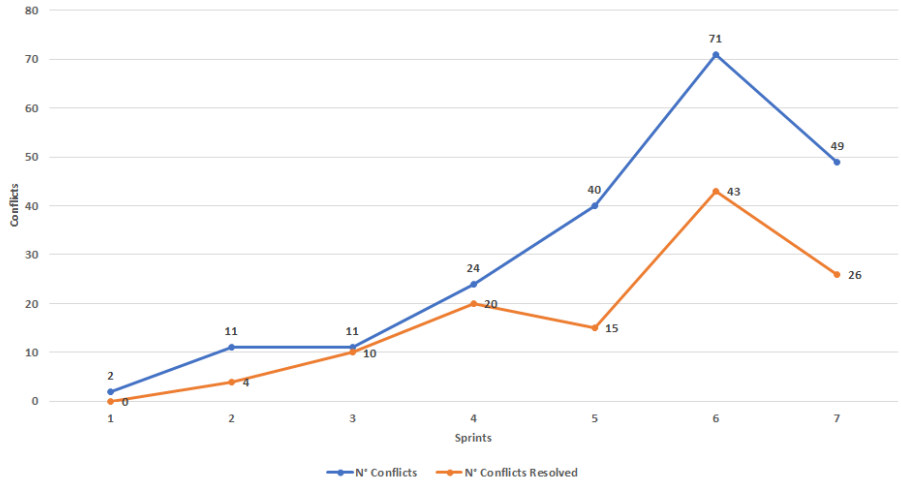

Fig. 4: Total conflicts vs. Conflicts resolved by Almost Rerere

Overall Almost Rerere proposed a resolution for 57% of conflicts occurred during development. The quality of the resolution depends on the intra-cluster similarity. Almost Rerere created 21 clusters for 121 different conflicts. 7 of those clusters have an intra-cluster similarity above 90%. By manual inspection, it was observed that for those clusters the CRR provided a good result directly applicable to solve the conflicts. In other 7 cases, clusters have intra-cluster similarity below 90% and the proposed resolutions required manual inspection to verify that they were syntactically and semantically correct. In the remaining cases, the cluster contained only 1 or 2 samples and Almost Rerere could not generate a CRR for such isolated cases. It was observed that when a cluster had few samples, the generated CRR was very sensitive to small variations, such as spaces. As the number of conflicts in a cluster increased, the tool was able to generalize the CRR by taking into account the possible variations.

5.2 Large project repositories

To evaluate the quality of the automatic resolution provided by Almost Rerere, it was necessary to know the actual resolutions committed by developers and use them as ground-truth. In [16], nine data sets based on Git repositories from active Java open-source projects were created by extracting all the differences between the sequential commits to the master branch. From six such repositories, we extracted all the single-line changes and used them as conflicts resolved by developers. For each single-line change, the original state was considered as the conflict and the after state as the resolution. Almost Rerere was fed with the content of each conflict file, to execute the cycle of resolving the conflict, adding it to the corresponding cluster and updating the generated CRR. The provided resolution for each conflict was compared to the ground-truth and the similarity index between them was saved.

Almost Rerere provided resolutions for a high number of conflicts in each repository: Ant 54%, Cobertura 70%, Eclipse SWT 59%, FitLibrary 54%, JGrapT 68%, JUnit 49%. Table 1 shows the statistics of the evaluated repositories. Overall Almost Rerere resolved 55,7% of conflicts.

Repository	**N° Conflicts**	**N° Cluster**	**N. Conflicts Resolved**	**% Resolved**
Ant	10500	1294	5667	53,97
Cobertura	1260	179	885	70,24
Eclipse SWT	1355	382	799	58,97
FitLibrary	4399	337	2371	53,90
JGraphT	3200	238	2135	66,72
JUnit	4424	388	2166	48,96
Total	**25138**	**2818**	**14023**	

Table 1: Total conflicts, Clusters and Resolved conflicts

To verify the quality of the generated resolutions, they were classified according to the Jaro-Winker similarity with the original resolution. Three intervals were considered: 100 - 90 % for which synthesized resolution was equal or almost identical to the original one; 89 - 80 % for which the synthesized resolution was close to the original one with only small variations; $\leq$ 79 %, for which the synthesized resolution was rather different from the original one and required the developer's intervention (see Table 2). Out of the 14.023 conflicts resolved, 65,8% of the resolutions had a similarity score with the original resolution exceeding 90%. This shows that Almost Rerere was able to synthesize an accurate resolution in most cases based on previously resolved similar conflicts. Table 3 classifies the clusters by their intra-cluster similarity. It can be observed that the clusters with less than 79% intra-cluster similarity account for almost 50% of the total clusters. They represent conflicts that are not common (occurred only once) or not similar to other conflicts. It was also observed that in trivial

Repository	**100 - 90 %**		**89 - 80 %**		**< 79 %**	
Ant	3717	65,59%	791	13,95%	1159	20,45%
Cobertura	577	65,19%	100	11,29%	208	23,50%
Eclipse SWT	567	70,96%	128	16,02%	104	13,01%
FitLibrary	1690	71,27%	434	18,30%	247	10,41%
JGraphT	1748	81,87%	221	10,35%	166	7,77%
JUnit	1470	67,86%	370	17,08%	326	15,05%
Total	**9229**		**2044**		**2210**	

Table 2: N° conflicts by resolution similarity intervals

Repository	**100 - 90 %**	**89 - 80 %**	**79 - 0 %**
Ant	127	485	682
Cobertura	27	33	119
Eclipse SWT	62	116	204
FitLibrary	48	132	157
JGraphT	26	86	126
JUnit	43	162	183
Total	**333**	**1014**	**1471**

Table 3: N° of cluster by intra-cluster similarity intervals

cases Almost Rerere could provide accurate resolutions even with very few examples, whereas in more complex cases it required more samples to generalize. For example, in the JGraphT repository, Almost Rerere created a cluster with 40 conflicts with intra-cluster similarity of 84%. Some examples of conflicts are:

```
[
  {
    "conflict": "public BreadthFirstIterator( Graph g ) {",
    "resolution": "public BreadthFirstIterator( Graph<V, E> g ) {"
  },
  {
    "conflict": "public UnmodifiableGraph( Graph g ) {",
    "resolution": "public UnmodifiableGraph( Graph<V, E> g ) {"
  },
  {
    "conflict": "public CycleDetector( DirectedGraph graph ) {",
    "resolution": "public CycleDetector( DirectedGraph<V, E> graph ) {"
  }
]
```

The common pattern is the addition of the generic expression *<V, E>*. In this case the learning process was simple and with only four samples Almost-Rerere converged to a CRR that provides a correct resolution:

```
[
  {
    "regex": "(h)( )",
    "replacement": "$1<V,$2E> "
  }
]
```

A different case is exemplified in the Cobertura repository, where a cluster contained the following conflicts:

```
[
  {
    "conflict": "return numberOfCoveredBranches;",
    "resolution": "return getRawCoverageData().getNumberOfCoveredBranches();"
  },
  {
    "conflict": "return numberOfCoveredLines;",
    "resolution": "return getRawCoverageData().getNumberOfCoveredLines();"
  },
  {
    "conflict": "return numberOfLines;",
    "resolution": "return getRawCoverageData().getNumberOfValidLines();"
  },
  {
    "conflict": "return numberOfBranches;",
    "resolution": "return getRawCoverageData().getNumberOfValidBranches();"
  }
]
```

In this case Almost Rerere, based on the first conflict, generated a CRR that transforms the name of the variable into a getter method respecting the Java notation. When the second conflict occurred, the CRR continued to work well. For the third conflict, the expression did not generated the expected result because the resolution added the word `Valid` to the name of the method. Almost Rerere integrated the ground truth of the third example into the cluster and generated a composite CRR, with different patterns for the two cases:

```
[
  {
    "regex": "(m*+)\w(umberOfCovered)(\w++)(;)",
    "replacement": "$1getRawCoverageData\(\)\.getN$2$3\(\)$4"
  },
  {
    "regex": "\w(\w\w\w\w\w\w\w)(\w+)",
    "replacement": "getRawCoverageData\(\)\.getN$1Valid$2\(\)"
  }
]
```

The composite CRR also worked when the fourth conflict occurred and provided an accurate resolution. This examples shows that Almost Rerere can adapt quickly when different unseen examples become available. Still it is sensitive to small changes when the number of samples available is small.

6 Conclusions

The paper describes an approach for the automatic resolution of conflicts during code integration on Git repositories. The approach is based on the synthesis of a search regular expression and a replacement expressions from previously resolved similar conflicts. A reference implementation, *Almost Rerere*, which extends the functionality of *Git Rerere*, was introduced, and the components in charge of executing the different steps of the approach, such as conflict clustering, regular expression generation and conflict resolution, were described. The proposed approach was evaluated in two use cases showing that it was able to resolve more than 55% of the observed conflicts. It was also shown, in the second use case, that more than 65% of the generated resolutions had a similarity score above 90%

with the ground truth. Future work will focus on improving the Cluster Manager by adding a dynamic re-clustering capability to keep cluster intra-similarity high, this would prevent the CRR and CC become outdated over time. It would also extend the approach for the detection and resolution of multi-line conflicts.

References

1. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proceedings of 2nd Working Conference on Reverse Engineering. pp. 86–95. IEEE (1995)
2. Bartoli, A., Lorenzo, A.D., Medvet, E., Tarlao, F.: Inference of regular expressions for text extraction from examples. IEEE Transactions on Knowledge and Data Engineering **28**(5), 1217–1230 (May 2016)
3. Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Automatic search-and-replace from examples with coevolutionary genetic programming. IEEE transactions on cybernetics (2019)
4. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). pp. 368–377. IEEE (1998)
5. Bernaschina, C., Comai, S., Fraternali, P.: Ifmledit.org: model driven rapid prototyping of mobile apps. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. pp. 207–208. IEEE Press (2017)
6. Bernaschina, C., Falzone, E., Fraternali, P., Herrera, S.: The virtual developer: Integrating code generation and manual development with conflict resolution. ACM Transactions on Software Engineering and Methodology (TOSEM) **28**(4), 20 (2019)
7. Brambilla, M., Fraternali, P.: Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML. Morgan Kaufmann (2014)
8. De Souza, C.R., Redmiles, D., Dourish, P.: 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. pp. 105–114. ACM (2003)
9. Ducasse, S., Rieger, M., Demeyer, S.: 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). pp. 109–118. IEEE (1999)
10. Hamza, J., Kunčak, V.: Minimal synthesis of string to string functions from examples. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 48–69. Springer (2019)
11. Jaro, M.A.: Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. Journal of the American Statistical Association **84**(406), 414–420 (1989)
12. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th international conference on Software Engineering. pp. 96–105. IEEE Computer Society (2007)

13. Johnson, S.C.: Hierarchical clustering schemes. Psychometrika **32**(3), 241–254 (1967)
14. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7), 654–670 (2002)
15. Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: 2006 13th Working Conference on Reverse Engineering. pp. 253–262. IEEE (2006)
16. Kreutzer, P., Dotzler, G., Ring, M., Eskofier, B.M., Philippsen, M.: Automatic clustering of code changes. In: Proceedings of the 13th Int. Conference on Mining Software Repositories. pp. 61–72. MSR '16, ACM, New York, NY, USA (2016)
17. Le Nguyen, H., Ignat, C.L.: An analysis of merge conflicts and resolutions in git-based open source projects. Computer Supported Cooperative Work (CSCW) **27**(3-6), 741–765 (2018)
18. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. vol. 10, pp. 707–710 (1966)
19. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: 2008 IEEE 24th International Conference on Data Engineering. pp. 257–266. IEEE (2008)
20. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on software Engineering **32**(3), 176–192 (2006)
21. Marcus, A., Maletic, J.I.: Identification of high-level concept clones in source code. In: Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001). pp. 107–114. IEEE (2001)
22. Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics. In: icsm. vol. 96, p. 244 (1996)
23. Meng, N., Hua, L., Kim, M., McKinley, K.S.: Does automated refactoring obviate systematic editing? In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. pp. 392–402. IEEE Press (2015)
24. Meng, N., Kim, M., McKinley, K.S.: Lase: locating and applying systematic edits by learning from examples. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 502–511. IEEE Press (2013)
25. Miller, R.C., Myers, B.A.: Lapis: Smart editing with text structure. In: CHI Extended Abstracts. pp. 496–497 (2002)
26. Nakatsu, N., Kambayashi, Y., Yajima, S.: A longest common subsequence algorithm suitable for similar text strings. Acta Informatica **18**(2), 171–179 (1982)
27. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of molecular biology **48**(3), 443–453 (1970)
28. Smith, T.F., Waterman, M.S., et al.: Identification of common molecular subsequences. Journal of molecular biology **147**(1), 195–197 (1981)
29. Tichy, W.F.: Rcs—a system for version control. Software: Practice and Experience **15**(7), 637–654 (1985)
30. Winkler, W.E.: String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. (1990)