POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Master of Science in Computer Science and Engineering

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

Supervisor: Prof. Piero Fraternali
Assistant Supervisor: Sergio Luis Herrera Gonzalez

Master Graduation Thesis
Mohammad Manan Tariq
877011

Academic Year 2018/2019

Abstract

Model-Driven Software Engineering provides tools to help developers in the creation process of complex applications. A developer, due to the continuous updates of the model and the generation of the code, must resolve many conflicts when integrating the generated and manually modified code. Several times the conflicts to be resolved are very similar to previously resolved conflicts. Our approach extends the work done on Almost-Git [1] and extends the Git Rerere tool, thus creating *Almost-Rerere*. The developed tool clusters similar conflicts using a hierarchical agglomerative clustering algorithm based on a similarity measure and synthesizes a conflict resolution based on those previously seen similar conflicts. The approach has been evaluated using it during the development of an application with model-driven development tools and using the history of large Git project repositories.

Sommario

Model-Driven Software Engineering fornisce strumenti per aiutare gli sviluppatori nel processo di creazione di applicazioni complesse. Uno sviluppatore, a causa dei continui aggiornamenti del modello e della generazione del codice, deve risolvere molti conflitti quando integra il codice generato e quello manualmente modificato. Spesso i conflitti da risolvere sono molto simili a quelli precedentemente risolti. Il nostro approccio estende il lavoro svolto su Almost-Git [1] ed estende lo strumento Git Rerere, creando così *Almost-Rerere*. Lo strumento sviluppato raggruppa i conflitti simili usando un algoritmo di raggruppamento agglomerativo gerarchico basato su una metrica di somiglianza e sintetizza una risoluzione del conflitto basata sui conflitti simili precedentemente visti. L'approccio è stato valutato utilizzandolo durante lo sviluppo di un'applicazione con gli strumenti di sviluppo basati su modelli e utilizzando la cronologia dei repository di grandi progetti Git.

Acknowledgements

First I would like to thank Prof. Piero Fraternali for giving me the opportunity to work on this interesting project. I would like to thank Sergio for his great support and advice during the development of the thesis work.

Especially I would like to thank all my family that has always supported me in difficult times and for all the sacrifices they made to give me the opportunity to complete my studies.

I would also like to thank all my friends I met during my studies. We have studied and spent many good times together. I will miss the moments we spent playing different games during breaks. I will always carry in my heart all the moments spent together. Thank you so much guys!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations and Background of the research

In recent years the technology sector has grown exponentially. IT companies to keep up with market changes are always looking for new technologies to reduce development costs. But at the same time, they want to increase the speed of software development without losing the quality of the final product. For this reason, more and more companies are resorting to the Model-Driven Software Engineering (MDSE) approaches which aim to increase productivity by simplifying the design process, application development process and optimizing system compatibility.

The software complexity increased in years, to overcome this problem, distributed development approaches were adopted. Software is developed by several development teams located in different locations. The parts of the software are shared between different teams through distributed development tools, this induces conflicts between the modified pieces of code that need to be manually resolved by one of the developers. When MDSE techniques are used this also induces conflicts between the automatically generated code and the human-written one. Many times developers resolved the similar conflicts over and over, the objective of this work is to provide tools that help developers to reduce the effort that they need to put in trivial tasks.

1.2 Approach of the thesis

This thesis project proposes and implements an approach that extends the capabilities of an open-source tool, Git Rerere, to automatically resolve conflict by synthesizing a resolution based on similar conflicts resolved in past iterations of the development process.

The algorithm identifies conflicts using a similarity metric. They are clustered together and a machine learning algorithm is applied to synthesize a general regular expression and replace expression to identify the small differences and automatically resolve the future similar conflicts.

The approach has been evaluated by using it during the development of an application under an agile approach and using model-driven development tools, and it has been evaluated using the history of large Git project repositories.

The contribution of this project can be summarized as the following:

- Git Rerere has been extended to resolve not only conflicts that have previously been resolved but to identify new conflicts with high similarity score to previously resolved and to synthesize the corresponding resolution based on those seen samples.

- Git Rerere has also been modified to extend its basic functionality, 2 configurable heuristics were developed to resolve conflicts originated on no-semantically representative sections of the code, e.g. conflicts cause by white-spaces or that occurred in comments, and to resolve previously resolve conflicts in files with more than 1 conflicting area.

- A hierarchical agglomerative clustering algorithm based on Jaro-Winkler String similarity has been developed to identify and group conflicts with a similarity score over a defined threshold.

The rest of the thesis is organized as follows, chapter 2 presents the recent research and works on related topics to contextualize the contributions of this project. Chapter 3 presents in detail the core concepts of the conflict resolution process and the proposed approach for automatic resolution. The chapter 4 describes the design and implementation of the components that were created for the automatic resolution process: identification of previously resolved conflicts, resolution synthesis, and resolution replacement, the chapter includes descriptions of

the Git Rerere process that were modified to integrate these components. Chapter 5 describes the experiment settings that were created to evaluate the tool on a Model-Driven development life-cycle and on large Git project repositories, it presents the results obtained from such evaluation. Finally, chapter 6 summarizes the conclusion obtained by the thesis project and gives an introduction to the work that will be done in the future to extend the capabilities of the tool.

Chapter 2

Related Work

In this chapter, we will survey approaches for Model-Driven Software Engineering. In particular, we will focus on the problem of integration of handwritten and generated code. In Section 2.1 we will explain the core concepts of MSDE and will survey the tools based on MSDE that have been presented in the last decade. In Section 2.2 we will survey the techniques used to integrate handwritten and generated code. In Section 2.3 we will survey the tools that allow teams to work concurrently on the same project. In Section 2.4 we will survey the techniques to merge the software and to resolve the conflicts. In Section 2.5 we will survey the state of the art to synthesis search and replace expression from examples.

2.1 Model Driven Software Engineering

Model-Driven Software Engineering (MDSE) uses the models as the main ingredients of software development. It is intended to reduce development effort by generating executable code from high-level models. The models are abstract representations of the system and they allow to represent the different aspects of the system. MDSE aims at improving the development process through automation [2].

The core concepts of MDSE are domain-specific modeling languages (DSMLs) and model transformations [3]. A domain-specific language (DSL) is a programming language or executable specification language for a particular domain and allow to express the concepts through appropriate notations and abstractions. [4]. DSMLs, such as IFML (Interaction Flow Modeling Language [5]), is specific for the model

domain. They support higher-level abstractions than general-purpose modelling languages, so they require less effort and fewer low-level details to specify a given system.

A model transformation performs a transformation between a source and a target model. There are two common model transformations: Model to Model (M2M) and Model to Text (M2T). The former transforms a model to another model and the latter generates textual artifacts from the model element. Transformation, defined at meta-model level, is applied to models that conform to meta-models. A meta-model is a precise definition of the constructs and rules needed for creating models. A meta-model has a high-level abstraction, so it does not contain low-level details [6]. The low-level details can be included in the final code using a template-based approach. An increasing number of companies are using code synthesis techniques to reduce development time and among them, template-based code generation is one of the most used [7].

In the last ten years, a great variety of tools based on Model-Driven Engineering (MDE) have been developed. *Sparx Enterprise Architect* [8] is an OMG UML based tool for designing and building business process modeling systems. *Metacase MetaEdit+* [9] is a purely graphical language tool that allows to build modelling tools and code generators without writing a line of code. A modelling language can be defined via meta models, constraints, and graphical symbols. Language concepts can be represented by different symbols in different diagrams types and elements from several languages can be used in one diagram [10]. *JetBrains Meta Programming System* (MPS) [11] is developed by JetBrains, it is an open-source projectional language workbench [12] that enable users to edit a projection, a visual or graphical representation of the data structure. MPS is based on a BaseLanguage which can be extended to define a new language.

Eclipse Modeling Project [13] provides a set of tools and frameworks, including Eclipse Modeling Framework [14], graphical modeling, textual modeling, and concrete modeling tools. *IFMLEditor.org* [15] is an online open-source tool for the rapid prototyping of web and mobile applications. The model of application is defined using IFML, the domain model is inferred from the IFML diagram, and actions are treated as abstract black-boxes. IFMLEditor.org is based on ALMOsT.js [1] transformation framework, which allows the developer to specify model

transformations with a rule-based extension of JavaScript [16].

2.2 Integration of handwritten and generated code

The main problem with model-driven approaches is that it is not possible to represent all features in the model, so it is necessary to add non-modelled features manually in code. The existing approaches share the idea that the handwritten and generated code must be managed separately. Specifically, the handwritten code must be structured, and possibly exposed to the code generator, in such a way that the generator does not overwrite the manual modifications.

In [17] the authors describe eight handwritten code integration mechanisms which can be adopted to reduce collisions. In [6] the authors present different methods such as protected areas, inclusion of code in models and templates. Protected areas approach is a simple mechanism to integrate handwritten and generated code. Developers must change the code only in these areas, any changes outside of these areas will be overwritten by the code generator. For example, the industry-standard transformation framework Acceleo [18] offers protected areas, which are preserved between executions of the model transformation.

In [19] the authors survey the approaches to manage change in the meta-model. They classify techniques for propagating meta-model changes not only to the models but also to the transformations. In [20] a tool is proposed to apply semi-automatically changes of the input meta-model to the transformation that depends on it, aiding the developer to co-evolve the meta-model and the transformation.

The paper [16] presents a model and code co-evolution approach that addresses such a problem a posteriori, using the standard collision detection capabilities of Version Control Systems to support the semi-automatic merge of handwritten and generated code. In chapter 3 we will describe this approach in details.

2.3 Tools and approaches for distributed development

In recent years, the software development process has become increasingly complex. Presently, software is developed by different teams that

may be in different sites [21]. Distributed software development is essential to develop the same product or service among globally distributed sites [22]. Developers, located in geographically separated areas, need to collaborate and share the source code. Version Control Systems (VCS) like Git [23], Mercurial [24] or Subversion (SVN) [25] provide tools to control changes over the source code and to manage conflict resolution. VCS allows developers to work on their local branch which maintains multiple workflows independent of each other. When the feature is complete, Developers can merge their local branch into the master branch. Merge could generate conflicts because two developers have changed the same lines of the code in a file. Some conflicts are resolved automatically by the VCS, for others the manual intervention of the developer is required.

In [26], the authors discussed six main advantages of distributed development for companies:

- Lower development costs due to, for example, lower salaries in other regions of the world
- Leveraging time-zone effectiveness, increase the number of daily working hours in a 'follow-the-sun' development model
- Access the most talented developers
- Cross-site modularization of development work
- Innovation and shared best practice
- Closer proximity to market and customer

One of the VCS most used is Git, developed by Linus Torvald to support the development of Linux kernel [27]. Over time it has become an important tool for distributed development. Git is written in C language, respect to other VCS it has best performance. Git utilizes a central repository and a series of local repositories. Local repositories are exact copies of the central repository complete with the entire history of changes. SVN, another popular tool for distributed development, is a centralized version control system. All the files are stored on a central server and developers commit their changes directly to the main repository. It is easy to use and understand but working on one central server means there is a single point of failure. If there is an error, it can destroy all builds. It is only accessible online.

Git with respect to SVN is preferred for a few reasons [28]: faster to commit; possibility to work and commit the changes on the local repository, and send the changes to a central repository when all the code is fixed; available offline.

2.4 Software merging and conflict resolution

Software merging is an essential aspect of the maintenance and evolution of large-scale software systems. It is assisted by VCS, which allows developers to work on a local branch, which are reconciled to produce a shared version. Over the years, a wide variety of different merge techniques has been proposed [29][16]. *Two-way merging* technique merge two versions of a software without considering the common ancestor from which both versions originated, instead *three-way merging* technique also use the common ancestor during the merge process. Software merging techniques at text level can be grouped into categories: *text-based* approaches consider a software artifact as a plain file and apply merging at the level of text lines; *syntactic* techniques also consider the syntax of the language and can avoid the production of unnecessary conflicts, e.g., those arising inside comments; *semantic* techniques consider the semantics of the language, which enables the detection of conflicts that a purely syntactic check would not identify; finally, *operation-based* techniques consider not only the artifacts but also the operations made by developers on them.

The survey work [30] provides an overview of the state-of-the-art of VCS dedicated to modelling artifacts. [31] exploits model differencing techniques to propagate the changes applied to a model to the artifacts depending on it, including those representing non-modelled features. The proposed approach requires specific transformations for all the possible types of model changes; for the data model, it can migrate the database schema and instance after a change, but for the presentation layer can only compute hints for the manual propagation of change. Commercial MDD tools, including Mendix [32], WebRatio [33], and Outsystems [34], offer functions for model versioning, visualization of model changes, and conflict resolution.

2.5 Automatic synthesis of search and replace expressions

Regular expressions are widely used in all type of tasks, not only in programming. However, writing a regular expression is not easy, for this reason, plenty of approaches for the automatic generation of regular expression have been proposed in the last years.

In [35], the algorithm requires input a set of examples and initial regular expression. The algorithm applies a series of transformations and stop when reaches a defined precision and recall. In [36] the algorithm does not require an initial regular expression. It extracts relevant patterns from a set of examples (phones numbers, invoice numbers) and evaluate the patterns using a fitting measure. The one that gives the best result is selected to generate a regular expression. In [37], the authors proposed a semi-supervised active learning algorithm to identify the patterns from textual data. It starts from a single example and produces a reduced form of a regular expression, but this is not completely automatic because it requires user intervention. Also [38] presents an active learning algorithm to generate regular expression, which requires human intervention to respond to membership queries about candidate expressions.

In last three decades, many approaches using genetic algorithms [39], [40] and genetic programming [41], [42], [43], [44], [45]. All these cited works does not consider the problem of synthesizing both a regular expression and a replacement expression. The authors of [43]-[45] presented a paper [46] in which they describe a completely automatic algorithm, *search-and-replace from examples*, that synthesize a regular expression & replacement from a set of examples. The authors developed a tool that can be tested online [47]. Our approach integrates this tool to synthesize a regular expressions & replacement from a set of conflicts and resolutions. In 4.1.2 we will describe in details the functionalities of algorithm search-and-replace.

Our approach extends the capabilities of a distributed development tool, Git, to automatically solves the conflicts by learning a search pattern and synthesizing a resolution from previously resolved similar conflicts. Our approach requires input a conflict and proposes to the user a possible resolution based on previous similar conflicts. Previously resolved conflicts are grouped using a hierarchical agglomerative

clustering algorithm based on Jaro-Winkler similarity.

Chapter 3

Proposed Approach

In this chapter, we will focus on the problem of automatic resolution of the conflicts. In Section 3.1 we describe an approach to integrate handwritten and generated code that has been presented in the paper [16]. In Section 3.2 we show how this approach is integrated into Git. Finally, in Section 3.3 we proposed an approach to resolve automatically conflicts when a developer submits his changes to the remote branch. Before proceeding, we introduce the notations use in 3.1:

- **Developer**: D_i denotes the i-th member of the development team.
- **Local/central code base**: C_i denotes the version of the code-base edited by developer D_i. C_C identifies the central code-base shared among developers.
- **Revision**: $R_{i,j}$ denotes the j-th revision of code-base C_i; it is the full textual artifact stored in C_i at a particular point in time. $R_{C,j}$ denotes a revision in the central code base.
- **Equivalence**: two revisions $R_{i,j}$ and $R_{m,n}$ are equivalent, if the content of the textual artifact stored in them is the same.
- **Difference**: the difference $R_{i,j} - R_{m,n}$ of two revisions is the set of changes that need to be applied to $R_{m,n}$ to produce $R_{i,j}$.
- **Delta**: the delta introduced by $R_{i,j}$ ($\Delta_{i,j} = R_{i,j} - R_{i,j-1}$) is the difference between $R_{i,j}$ and the previous revision $R_{i,j-1}$.
- **Manual Revision**: $R^M_{i,j}$ denotes revision resulting from a manual change.

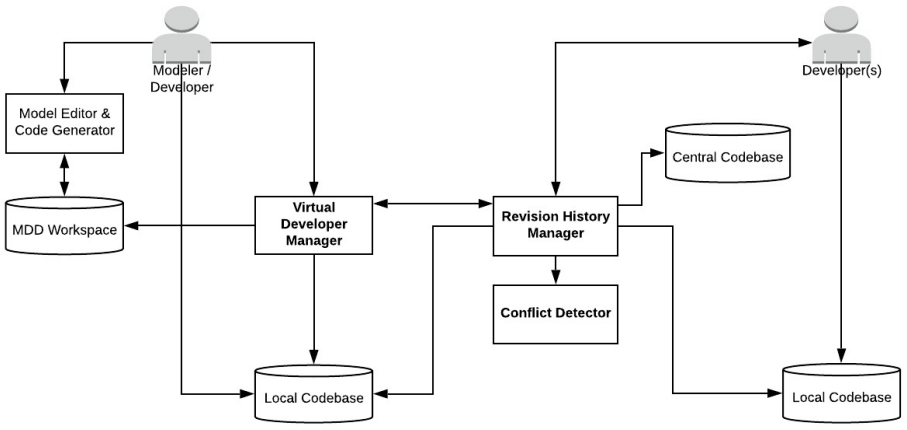

Figure 3.1: Architecture supporting the Vitrual Developer work-flow

- **Generated Revision**: $R^G_{i,j}$ denotes revision containing generated artifacts.
- **Resolution Revision**: $R^R_{i,j}$ denotes revision resulting from conflict resolution.

3.1 Integration of handwritten and generated code: The Virtual Developer workflow

Model-to-text transformations generate textual artifacts from models and each time regenerate the whole textual artifact. At every generation, it would be as if an update on the entire artifact is checked in to the repository regardless of the past history of the code. Submitting a generated revision without precautions would "forget" the conflict resolution steps done in precedence and make the same sources of conflict arise again at each code generation. But if the human developer has modified the code generated from a model element E and the subsequent generation applies to a model in which E has not been updated, then the manual modification is still valid and should not be considered as a source of conflict. To make the behaviour of the code generator more similar to that of the human developer, redundant information can be inserted in the code-base history, simulating the incremental production of code by the model transformation.

A model-to-text transformation can be considered as a *Virtual De-*

veloper. Treating the model-to-text transformation as an additional developer potentially simplifies the management of manual updates in the forward engineering MDD life cycle. Tools and methodologies that are normally applied for conflict resolution among developers can be applied to both human and virtual developers. This model and co-evolution approach was presented in the paper [16]. Figure 3.1 shows the whole architecture described in the cited paper.

Multiple human developers and a single Virtual developer can work in parallel, in a work-flow such as the following:

1. C_C contains n commits, the latest commits $R^G_{C,n}$ is generated.
2. A human developer D_H creates a local copy C_H of the remote branch C_C.
3. D_H introduces a new feature by applying changes to $R^G_{H,n}$, creating a new commit $R^M_{H,n+1}$.
4. D_H submits the new commit to the remote branch generating a new commit $R^M_{H,n+1} \rightarrow C_C = R^M_{C,n+1}$. The submission is accepted because $R_{C,n} = R_{H,n}$.
5. D_H deletes her local branch C_H returning to the initial state. Her work is safely stored in C_C.
6. The model is updated and the transformation is ready to be executed.
7. The Virtual Developer D_V creates a local copy C_V of the remote branch C_C *up to the latest generated commit*: $\forall_{j \in \{1..k\}}(R_{V,j} = R_{C,j})$, where k is the index of the last generated commit (k = n in the current example).In this way, the Virtual Developer aligns its local branch to the status that reflects the previous version of the model.
8. D_V executes the transformation and stores the result into C_V generating a new commit $R^G_{V,n+1}$, which is a replacement of the entire textual artifact.
9. D_V tries to submit $R^G_{V,n+1}$ to the remote branch. The operation is rejected because $R^G_{V,n}$ is not equivalent to $R^M_{C,n+1}$, due to the intervening manual update of D_H.

10. D_V solves the conflict between $R^M_{C,n+1}$ and C_V generating $R^M_{C,n+1} \rightarrow C_V = R^R_{V,n+2}$. In this step, the $\triangle^G_{V,n+1}$ used to identify the modifications introduced by the latest round of code generation, which simplifies, and even automates in some cases, the identification and resolution of collisions with the manual modifications of the code.

11. In order to safely store the latest generated commit in the remote branch for future alignment, D_V forces the submission of $R^G_{V,n+1}$ to C_C, generating $R^G_{C,n+2}$.

12. D_V submits the conflict resolution $R^R_{V,n+2}$ to the remote branch, generating a new shared commit $R^R_{V,n+2} \rightarrow C_C = R^R_{C,n+3}$. The submission is accepted because $R^G_{C,n+2}$ is equivalent to $R^G_{V,n+1}$. It is important to notice that $\triangle_{C,n+3}$ is identical to $\triangle_{V,n+2}$, due to the previous forced submission, which saved in the remote branch also the latest generated revision.

13. D_V deletes its local branch C_V returning to the initial state. Its work is safely stored in C_C.

All steps of Virtual Developer workflow are show in Figure 3.2. The remote branch contains a twofold sequence of commits: automatically generated ($R^G_{C,j}$) and conflict-resolved ($R^R_{C,j+1}$). The human developer always works on the latest commit, whereas the Virtual Developer is always out-of-sync and applies changes on the latest generated commit $R^G_{C,j}$.

The limits of Almost-Git are the following:

1. it does not support multiple Virtual Developers and conflict resolution at the model level;

2. it works only on Git repositories;

3. item cannot automatically resolve conflicts on the code manually modified by the developer

Our approach overcomes the last limit of Almost-Git and automatically resolve conflict by synthesizing a resolution based on previously resolved conflict with a high similarity score.

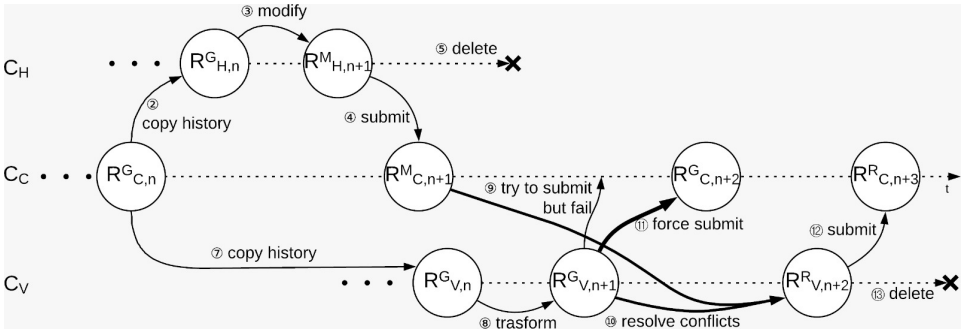

Figure 3.2: Vitural Developer workflow

3.2 Conflict resolution with Almost-Git

Almost-Git [48] is a control manager for the Virtual Developer explained in 3.1, it maps the operations required to handle a submission of the Code Generator into calls to the Revision History Manager interface. The Virtual Developer concepts are mapped to Git primitives as follows [16]:

- *code-bases* are mapped to Git *branches*
- *revisions* are mapped to Git *commits*.
- the act of copying the central code-base is mapped to the *clone* or *branch* Git operations, depending on the location of the central code-base; the former in the case of a centralized repository, the latter in the case of a local branch.
- *submission* is mapped to the Git *push* operation, which copies commits from the current branch to another local or remote one. The *push* operation may fail if the latest commit in the remote branch (i.e., the HEAD of the branch) is not identified in the local branch.
- *collision resolution* is mapped to the pull operation in Git.

The evolution process of Virtual Developer can be controlled with the following commands:

- *almost-git init*: initialize the repository and sets the status to READY

- *almost-git evolve /new-generated-code*: starts the evolution process. If a conflict arises during execution then the status is set to CONFLICTED and the human developer will be asked manually solve the conflicts and continue the evolution process.
- *almost-git evolve –continue*: continues the evolution process after the human developer has resolved manually the conflicts.
- *almost-git evolve –abort*: abort the execution process and restore the repository status to the initial state

Almost-Git is built on top of the Git, and it allows developers to preserve the parts of code that have been changed manually.

3.3 Conflict resolution with *Almost-Rerere*

Git Rerere stands for **RE**use **RE**corded **RE**solution, which means that it automatically records a conflict and its resolution so that the next time the same conflict occurs again Git can automatically resolve it using the previously recorded solution. Git Rerere can be useful in several cases. For example, on a long-lived feature branch where refactoring of methods tends to be common and the developer might need to execute several integration testing cycles before the release of the feature.

The Rerere is a useful tool, but its implementation has some problems that limit its potential:

- The Rerere mechanism relies heavily on the HASH of the conflict areas, any change that alters the hash (like a space character) will prevent Rerere from finding the recorded solution. A snapshot's content is used only if its conflict area context is preserved, as is usual for merge conflicts. Consequently, Rerere will not be able to locate the *hash* directory to resolve the conflict.
- The Rerere support multiple conflicts in a file, but if in a file there are two conflicts and Rerere has the solution for only one of them, then it is not able to apply the previous solution to the conflict because it generates a single hash per file. As there is a new conflict, the hash changes. Consequently, Rerere will not be able to locate the *hash* directory to resolve the conflict.

Our approach, *Almost-Rerere*, aims at overcoming these limitations and help developers to resolve the conflicts when Rerere fails to apply the previous solution. *Almost-Rerere* does not depend on the HASH so it is not affected by the changing in a conflict area. *Almost-Rerere* records conflict and its resolution, then use it to learn a search pattern that can be applied to all the similar conflicts.

The approach consists of extending the current functionality of Rerere: to resolve conflicts originated on no-semantically representative sections of the code, e.g. conflicts cause by white-spaces or that occurred in comments, and to resolve previously resolve conflicts in files with more than 1 conflicting area; to synthesize a regular expression and a replacement from a group of similar conflicts. The first functionality is trivial, we updated the pre-image and post-image by adding the section of codes that generate conflicts.

In this work, we concentrate on the development of the second feature to automatically resolves the conflicts. We cluster the similar conflicts and their resolution using similarity metrics. These clusters are given in input to genetic algorithm for automatic synthesis of a regular expression and a replacement expression. The regular expression and the replacement are applied to a conflict and the result is proposed to the user as a possible solution for the conflict. Initially, the generated regular expressions & replacements are very sensitive to small changes because the number of conflicts in a cluster are very small. But as the conflicts increase the regular expressions & replacements become more precise. In chapter 4 we describe the whole architecture of *Almost-Rerere*.

3.3.1 Git Rerere implementation

Git Rerere can be enabled globally when a repository is initialized the *.git/rr-cache* directory is created automatically, or locally by manually creating the *.git/rr-cache* directory in a specific repository. Rerere saves a pre-image and post-image of the file that has conflicts. Pre-image is a before-state of the file with conflicts and Post-image is, instead, after-state of the file once conflicts have been resolved. Figure 3.3 shows how *.git/rr-cache* is organized. The following code shows how a pre-image file looks like. The markers <<<<<<< and >>>>>>> delimits the conflict area, and the marker ======= separates the user local changes from the remote changes.

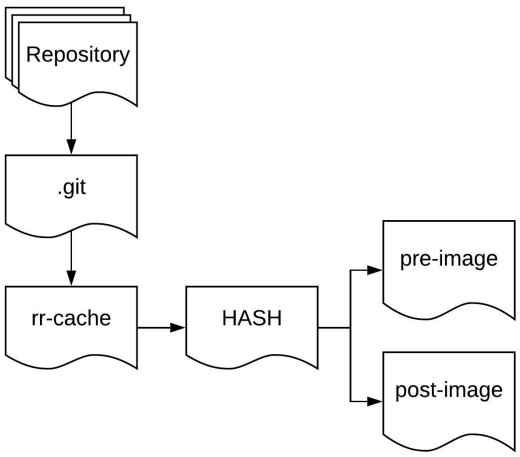

Figure 3.3: Rerere directory

```
<h3>List</h3>
<table class="table table-hover table-condensed">
<!-- Conflict area Start -->
<<<<<<< <!-- Start Marker -->
    <thead class = "HEader"> <!-- Local Branch -->
======= <!-- Separate Marker -->
    <thead class= "header">  <!-- Remote Branch -->
>>>>>>> <!-- End Marker -->
<!-- Conflict area End -->
        <tr>
            <th>#</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Username</th>
        </tr>
    </thead>
<!-- Conflict area Start -->
<<<<<<<
    <tbody data-bind="foreach: items" class = "BODY">
=======
    <tbody data-bind="foreach: items" class= "body">
>>>>>>>
<!-- Conflict area End -->
        <tr data-bind="click: $parent.select">
            <td data-bind="text: id"></td>
            <td data-bind="text: $data['First Name']" style="white-space:
                pre-wrap;"></td>
            <td data-bind="text: $data['Last Name']" style="white-space:
                pre-wrap;"></td>
```

```
28              <td data-bind="text: $data['Username']" style="white-space:
                    pre-wrap;"></td>
29          </tr>
30      </tbody>
31  </table>
```

The post-image code corresponding to the pre-image code is following:

```
1   <h3>List</h3>
2   <table class="table table-hover table-condensed">
3       <thead class= "header">
4           <tr>
5               <th>#</th>
6               <th>First Name</th>
7               <th>Last Name</th>
8               <th>Username</th>
9           </tr>
10      </thead>
11      <tbody data-bind="foreach: items" class= "body">
12          <tr data-bind="click: $parent.select">
13              <td data-bind="text: id"></td>
14              <td data-bind="text: $data['First Name']" style="white-space:
                    pre-wrap;"></td>
15              <td data-bind="text: $data['Last Name']" style="white-space:
                    pre-wrap;"></td>
16              <td data-bind="text: $data['Username']" style="white-space:
                    pre-wrap;"></td>
17          </tr>
18      </tbody>
19  </table>
```

Rerere is invoked automatically by different git commands such as: *merge*, *rebase*, *cherry-pick*, *stash apply/pop*, *checkout –m*. Git Rerere has several commands that allow it to interact with its working state:

- **clear**: reset the metadata used by rerere if a merge resolution is to be aborted.
- **forget <pathspec>**: reset the conflict resolutions which Rerere has recorded for the current conflict in <pathspec>.
- **diff**: display diffs for the current state of the resolution. It is useful for tracking what has changed while the user is resolving conflicts.

- **status**: print paths with conflicts whose merge resolution Rerere will record.
- **remaining**: print paths with conflicts that have not been auto-resolved by Rerere. This includes paths whose resolutions cannot be tracked by Rerere, such as conflicting submodules.
- **gc**: prune records of conflicted merges that occurred a long time ago. By default, unresolved conflicts older than 15 days and resolved conflicts older than 60 days are pruned. These defaults are controlled via the *gc.rerereUnresolved* and *gc.rerereResolved* configuration variables respectively.

To better illustrate the resolution process of Rerere, a typical use case of the tool is presented in the following:

A developer is working on a feature branch, his changes may overlap an area that the master branch touched, so he may want to perform integration tests with the latest master, even before feature branch is ready to be pushed upstream.

1. Developer execute *git merge master* command to get the changes of the master branch into the feature branch.
2. Git tries to merge automatically but it fails due to overlap of changes in a file, so *git merge* calls automatically Rerere.
3. Rerere verifies if there are any conflicts in the file, for each conflict area found it generates a *hash*.
4. Rerere search for the *.git/rr-cache/hash* directory, if the directory does not exist, it is created.
5. Rerere saves the pre-image inside it. After that Rerere terminates and Git asks developer to resolve manually the conflicts.
6. Developer resolves the conflicts and commits the changes, *git commit -m "Message"*.
7. Steps 3 and 4 are repeated.
8. Rerere finds the path *.git/rr-cache/hash* containing the pre-image. It saves the post-image of the file inside the *hash* directory.

   Developer after completing all the tests throw away the merge by executing *git reset –hard HEAD* and continue to work on the

feature branch. When the branch is ready to be pushed upstream, developer aligns the feature branch to the master.

9. Steps 1, 2, 3 and 4 are repeated.
10. Rerere finds the path *.git/rr-cache/hash* containing both pre-image and post-image files.
11. Rerere resolves the conflict performing a three-way merge between the pre-image, current state and post-image.
12. User commits and push all the changes to the remote master branch, *git push origin master*.

Figure 3.4 shows the flow diagram of the automatic resolution mechanism of Rerere.

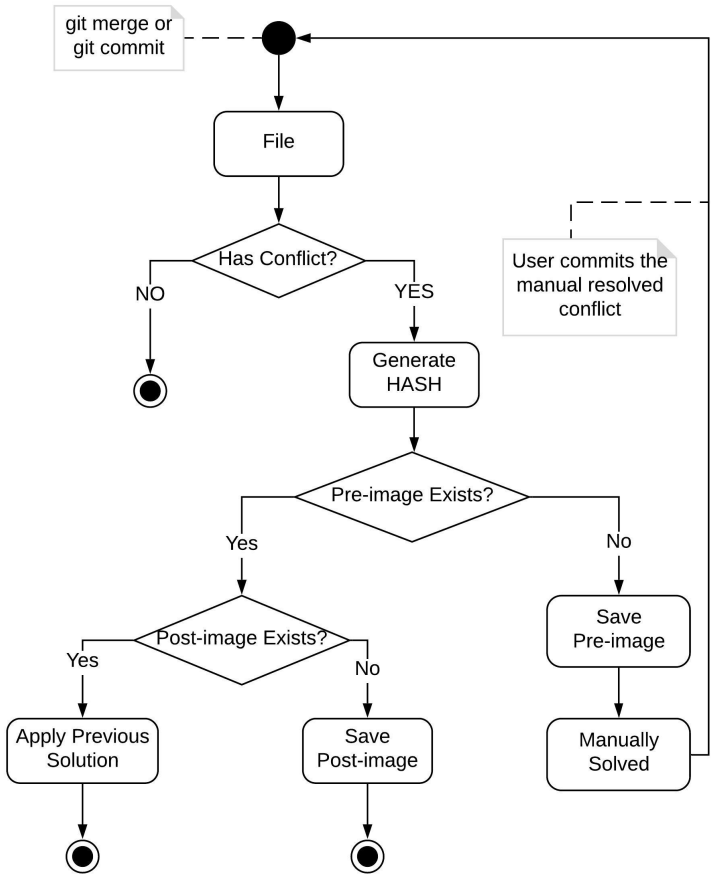

Figure 3.4: Git Rerere automatic resolution mechanism Diagram

Chapter 4

Implementation

In this chapter, we will illustrate the architecture of *Amost-Rerere* and for each component will show implementation details.

4.1 The *Almost-Rerere* architecture

Almost-Rerere architecture is shown in Figure 4.1. It is composed of three main components: conflict clustering, regular expression & replacement generator and conflict resolver.

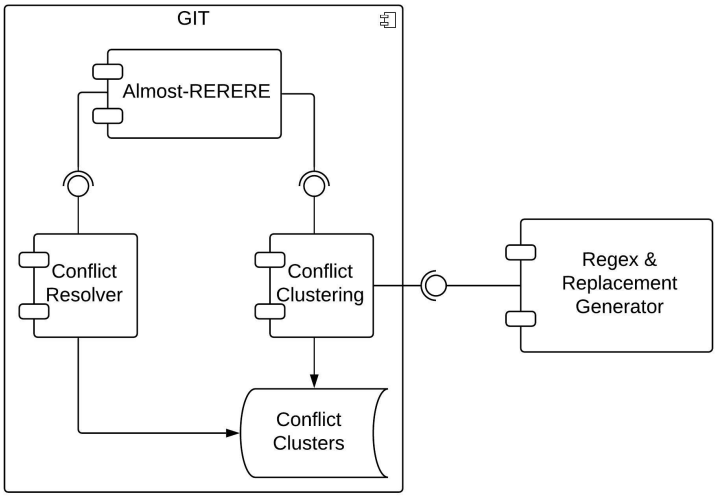

Figure 4.1: Almost-Rerere architecture

Conflict clustering and conflict resolver are developed in C language, and they are part of Almost-Rerere. Regular expression & replacement

generator [46], instead, is an external tool written in Java that is used by Almost-Rerere to generate regular expression & replacement. Now we will describe each component in detail.

4.1.1 Conflict Clustering

Clustering is the task of grouping a set of similar data in the same group called cluster. Different algorithms differ significantly in their understanding of what constitutes a cluster and how to efficiently find them, but none of them is better than the other. The best clustering algorithm is the one that best fits our data set.

Almost-Rerere clusters the conflicts using the hierarchical clustering [49] that group the objects in clusters based on their similarity. The result of hierarchical clustering is a set of clusters, where each cluster is distinct from each other cluster, and the objects within each cluster are broadly similar to each other. Hierarchical clustering starts by treating each object as a separate cluster. Then, it identifies and merges the two most similar clusters. These steps are repeated until all the clusters are merged. Hierarchical clustering uses a similarity measure to identify if two clusters or objects are similar. *Almost-rerere* use *Jaro-Winkler* similarity measure. Jaro-Winkler is a modification of Jaro.

The Jaro similarity of two string $s1$ and $s2$ is [50]:

$$sim_j = \begin{cases} \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{|m|}) & \text{m} > 0 \\ 0 & \text{m} \leq 0 \end{cases}$$

where:

- $|s_i|$ is the length of the string s_i
- m is the number of matching characters
- t is half the number of transpositions

The two strings, $s1$ and $s2$, are compared character by character. Two characters are considered *matching* only if they are the same and have a maximum distance of $\frac{max(|s_1|,|s_2|)}{2} - 1$. The number of matching character, but in different sequence order, divided by 2 defines the number of transpositions.

The Jaro-Winkler modified this algorithm to support the idea that differences near the start of the string are more significant than differences near the end of the string. The Jaro-Winkler similarity of two

	Average Similarity
Jaro-Winkler	0,95
Jaro	0,92
Cosine	0,91
Sorensen	0,91
Damerau	0,88
LCSS	0,88
Levenshtein	0,87
Jaccard	0,85
Hamming	0,68
LCSSTR	0,64

Table 4.1: Average similarity score over 200 samples

string $s1$ and $s2$ is [50]:

$$sim_w = sim_j + l * p * (1 - sim_j)$$

where:

- sim_j is the Jaro similarity of $s1$ and $s2$
- l is the length of common prefix at the start of the string up to a maximum l_{bound}
- p is a scaling factor for how much the similarity measure is adjusted upwards for having common prefixes

The standard values of l_{bound} and p ,as defined in Winkler's work [51], are respectively 4 and 0.1. The Jaro-Winkler distance is defined as $d_w = 1 - sim_w$.

Jaro and Jaro-Winkler measures are suited for comparing short strings. We evaluated different similarity measures on a set of strings that were to be considered similar to each other. We compared each string pairs, obtaining a similarity score. We have a data set of about 200 pairs, Jaro-Winkler has the highest similarity score in 80% of cases. Table 4.1 report average similarity score of different similarity measures.

Almost-Rerere compares a new conflict with all clusters one by one. For each cluster, it computes an average Jaro-Winkler's similarity score. After different tests, we set a threshold at 0.80. If there is more than

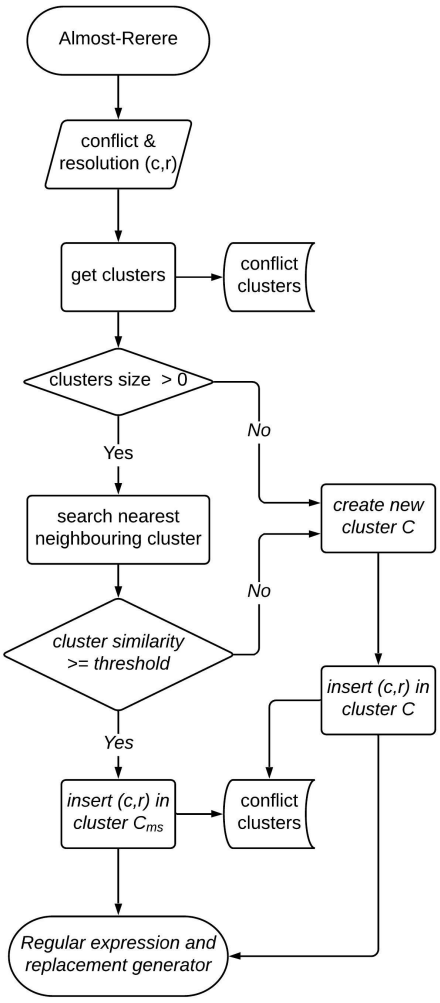

Figure 4.2: Almost-Rerere Conflict Clustering

one cluster that is above the threshold then the cluster with the highest similarity score is selected. The conflict and its resolution is inserted in the cluster which have maximum similarity score, only if it is over a threshold. If no cluster is found then a new cluster is created. A detailed flow of conflict clustering is shown in Figure 4.2.

An example of conflict clusters data set is shown below. Each cluster has a unique id and contains an array of objects. The object is composed of a conflict and a resolution.

```
{
    "1": [
        {
            "conflict": "<form class=\"blue\">",
            "resolution": "<form class=\"red\">"
        },
        {
            "conflict": "<form class = \"blue\">",
            "resolution": "<form class =\"red\">"
        },
        {
            "conflict": "<form class = \"blue\">",
            "resolution": "<form class =\"red\" >"
        }
    ],
    "2": [
        {
            "conflict": "<div class =\"row\">",
            "resolution": "<div class = \"row\">"
        },
        {
            "conflict": "<div class=\"space\"></div>",
            "resolution": "<div class = \"space\"></div>"
        },
        {
            "conflict": "<div class =\"space\" ></div>",
            "resolution": "<div class=\"space\"></div>"
        }
    ]
}
```

4.1.2 Regular expression and replacement generator

Almost-Rerere resolves the conflict using regular expression & replacement which are generated by an external tool, *Automatic Search-and-Replace From Examples* [46], written in Java.

The Search-and-Replace algorithm takes as input a series of examples consisting of pairs describing the original string and the desired modified string, in our case the conflict and its resolution. Starting from these examples, a search pattern and a replacement expression is generated: the former is a regular expression which describes both the strings to be replaced and their portion to be reused in the latter, which describes how to build the modified strings.

The algorithm is based on Genetic Programming (GP), a type of

an Evolutionary Algorithm (EA) inspired by biological evolution such as reproduction, mutation, recombination, and selection. The best solution is chosen based on a fitness function. The main steps of the algorithms are:

1. Initially generates a random population of individuals for search pattern and replacement.
2. Evaluate the generated population using a fitness function.
3. Repeat the following regeneration steps until termination:
   (a) Select the best-fit individuals (Parents).
   (b) Breed new individuals through crossover and mutation operations to generate a new population of search pattern and replacement.
   (c) Evaluate the individual fitness of new individuals.
   (d) Replace least-fit population with new individuals.

The tool has been modified to read our conflict clusters data set and to output a JSON file that has regular expression & replacement for each cluster. Almost-Rerere executes the Search-and-Replace tool, passing as parameters the ids of clusters that have been updated when user made a commit. Search-and-Replace tool, for each cluster, reads all the conflicts and the corresponding resolutions and constructs a set of data by dividing examples for training, validation and testing.

The algorithm needs at least three examples, if a cluster does not have three examples, the same sample is used for training, validation and testing. After every execution, we verify if the generated regular expression & replacement is valid, otherwise, the algorithm is executed another time. The generated regular expression & replacement could depend on the order of conflicts, to make it independent the algorithm is executed multiple times. If the generated regular expressions & replacements are same then the only one is written on the file, otherwise the expressions are used with an *or* logical connection. The algorithm terminates when a predefined number of generation has been executed or the best search-and-replacement expression is found.

The algorithm may take several minutes to run the clusters that have been updated, for which it runs on a new background thread. Figure 4.3 shows a detailed flow of Regex & Replacement Generator.

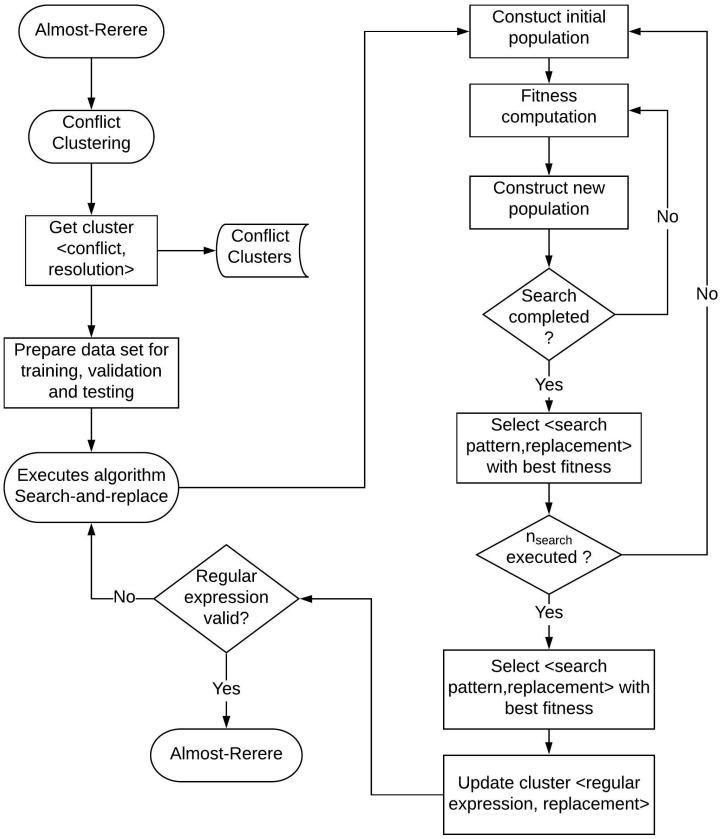

Figure 4.3: Almost-Rerere Regex & Replacement Generator

4.1.3 Conflict resolver

Conflict resolver has a structure very similar to conflict clustering. This component has the responsibility to resolve a conflict. Figure 4.4 shows how *Almost-Rerere* resolves a conflict.

Each cluster can have one or two regular expression & replacement. A cluster that has the highest similarity with respect to other clusters, is the closest to the conflict. The regular expression & replacement of the closest cluster is applied to the conflict, and the result is showed to the user. If there are two regular expressions & replacements then both are applied to the conflict. The two resolutions are compared using Jaro-Winkler similarity, the one that has the highest similarity score is returned to the user as the best possible solution. The regular expression & replacement for each cluster are stored in a JSON file, an example is shown below.

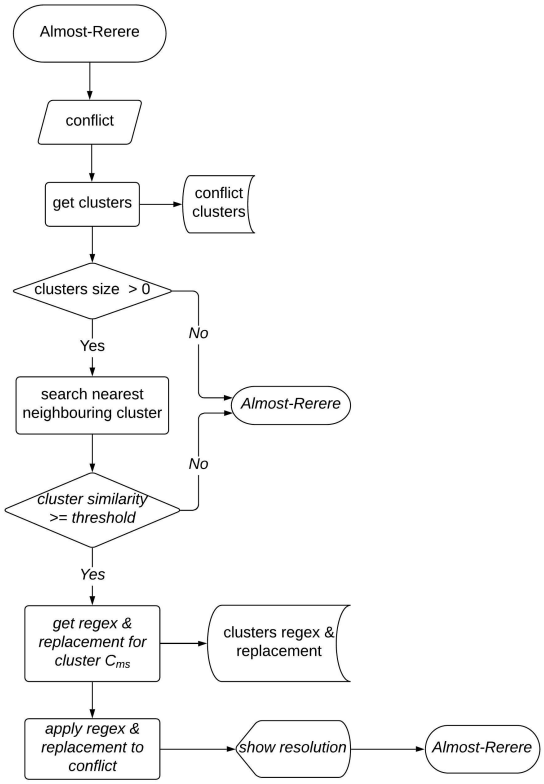

Figure 4.4: Almost-Rerere Conflict Resolver

```
{
    "1": [
        {
            "regex": "[^7]\"[^;]\\w*+",
            "replacement": "\"red"
        },
        {
            "regex": "(?:([^2])[^_]u(e)(\"))++",
            "replacement": "red$3 "
        }
    ],
    "2": [
        {
            "regex": "(?=\\))",
            "replacement": "s"
        }
    ],
    "3": [
        {
```

```
            "regex": " = ",
            "replacement": "="
        }
    ]
}
```

4.2 Git Integrations

Almost-Rerere is invoked automatically by *git merge* and *git commit* commands. It operates in two different cases: In first case, *Almost-Rerere* tries to resolve the conflicts using regular expressions & replacements and proposed to the developer a possible solution. In the second case, *Almost-Rerere* add conflicts in the clusters and updates the regular expressions & replacements for each cluster. The workflow for two cases are described below.

Figure 4.5 shows the flow of *Almost-Rerere* in first case. The *check_ conflict_ suggestion()* function reads the file with conflicts and extracts the conflict area. For each conflict line in the conflict area, the *regex_ repalce_ suggestion()* function computes a similarity measure between the conflict and the clusters. The conflict fit into the cluster that has the highest similarity compared to other clusters. The *apply_ regex_ replacement()* gets the regular expression & the replacement for the chosen cluster, and apply it to the conflict. The string obtained is returned to the developer as a possible solution for the conflict.

Figure 4.6, instead, shows the flow of *Almost-Rerere* in the second case, when a developer commits manually resolved conflicts. The *write_json_ conflict_ index()* function checks if the pair, conflict and its resolution is already present in a cluster. If not, it computes a similarity measure between the conflict and clusters. The pair is inserted in the cluster for which the similarity measure is maximum. After the conflicts have been added into a cluster, *Almost-Rerere* executes the external tool explained in 4.1.2 which updates the regular expression & the replacement for each modified cluster.

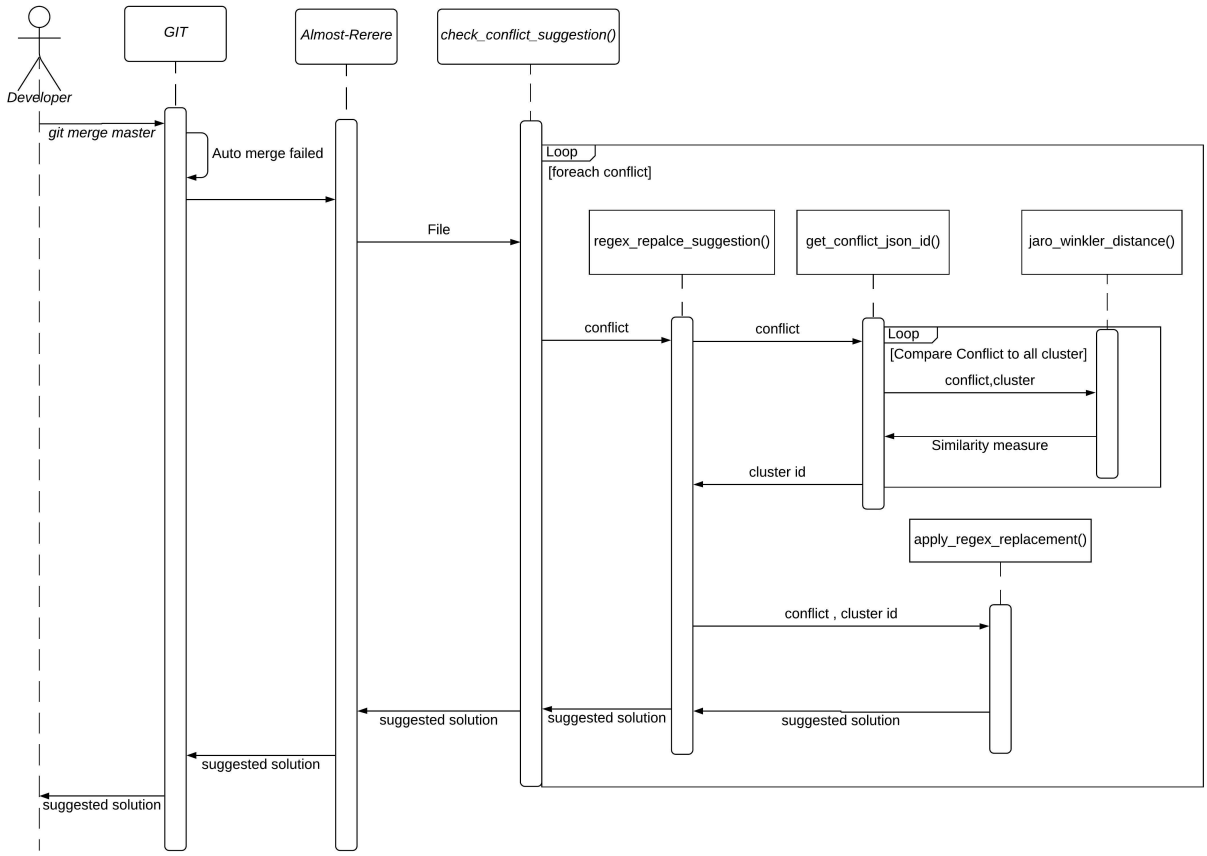

Figure 4.5: Almost-Rerere Git Merge sequence diagram

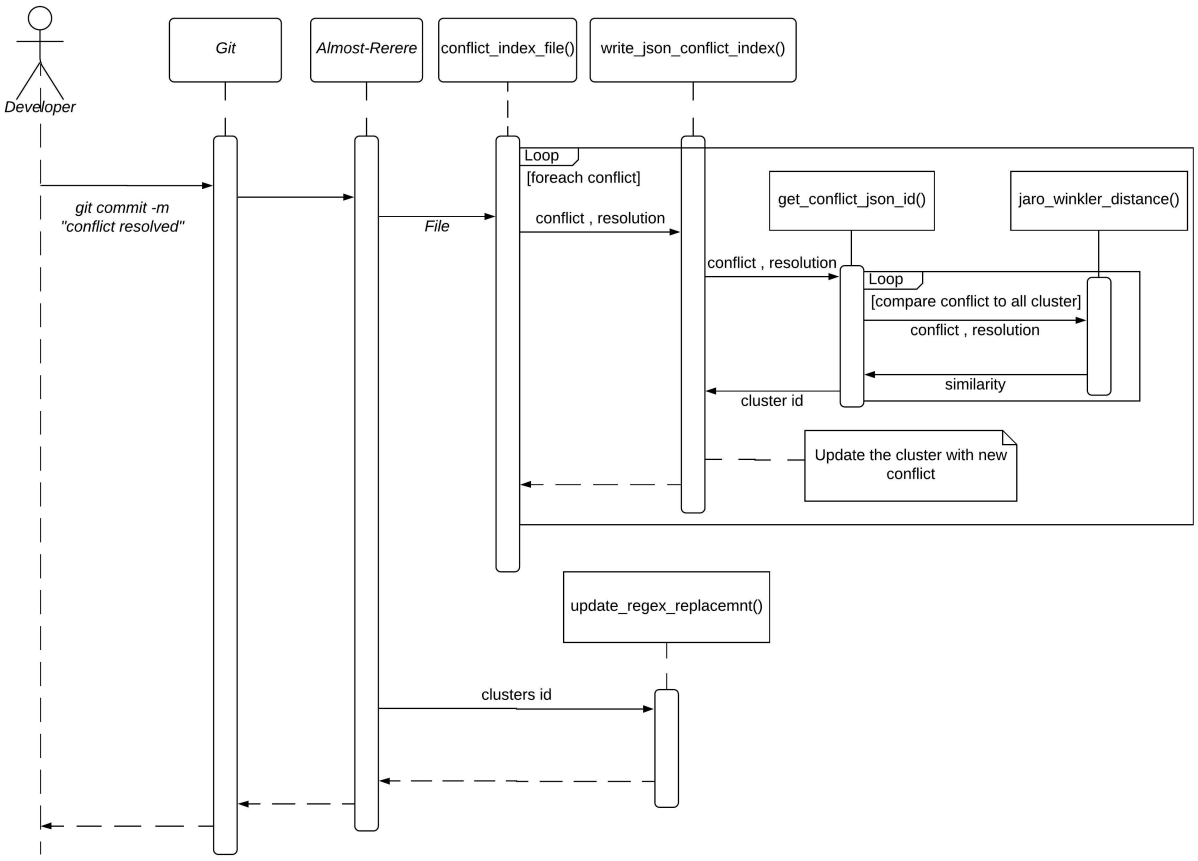

Figure 4.6: Almost-Rerere Git Commit sequence diagram

Chapter 5

Evaluation

We evaluated *Almost-Rerere* on two case studies. In the first case, we develop a web-based Crowd-Sourcing platform using a Model driven development tool and following an Agile development approach. In the second case, we evaluate our approach on large git repositories of long run open-source projects.

5.1 Conflict resolution results for the integration of handwritten and generated code

The application is developed using IFMLEdit.org [15]. IFMLEdit.org is an online tool for the rapid prototyping of web and mobile applications [52]. It is based on Interaction Flow Modeling Language (IFML) [5]. IFMLEdit.org allows developers to design the model of web or mobile applications, using a graphical interface. The following aspects of an interactive application can be defined in the tool:

- The view structure and content: the view content is expressed through two classes of ViewElements: *ViewContainers*, elements for representing the nested structure of the interface, and *ViewComponents*, elements for content display and data entry. *ViewComponents* have a *ContentBinding*, which expresses the link to the data source.
- The events: the occurrences that affect the state of the user interface, which can be produced by the user interaction, the application, or an external system.

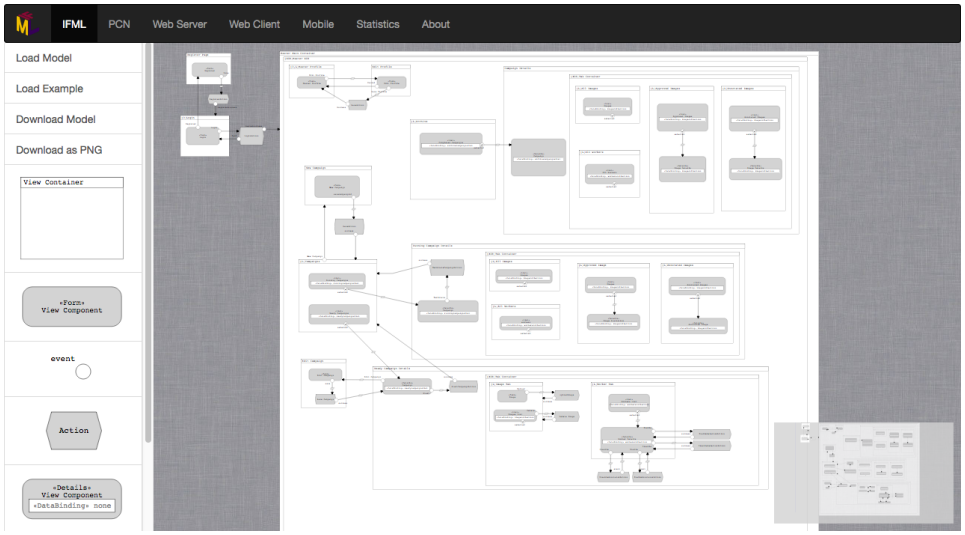

Figure 5.1: IFMLEdit.org online tool

- The event transitions: the consequences of an event on the user interface, which can be the change of the *ViewContainer*, the update of the content on display, the triggering of an action, or a mix of these effects. *Actions* are represented as black boxes.

- The parameter binding: the input-output dependencies between *ViewElements* and *Actions*.

The development process of the application is divided into seven sprints. At each sprint, we apply changes to the model or add a manual modification to code. We developed the application with the aim of creating conflicts. We simulated that two developers work on the development of the platform. The main repository is on master branch, each developer work on his own branch. The details of seven sprints are shown in the table 5.1.

In sprint from 1 to 6, both developers update the model with new content. Both developers make the same changes, but with little differences, to the templates of IFMLEdit.org. When Developers generate the code, the changes will propagate to all the pages where the modified templates are used. In sprint 7 the changes are made directly on the generated code. Both developers commit changes to their local branch. The main goal of these changes is to generate many conflicts when the code is committed to the main repository. In fact, when the local branches are merged into the main repository, it generates

Sprint	Description
1	Create Login and Register page: A User can create an account and login into the application.
2	Create User Profile page and Edit Profile page: A User can access to profile and change the account information (Name,Surname,password).
3	Create New Campaign page and List Campaign page: A User can create a new campaign and visualize list of all campaigns.
4	Create Ready Campaign Details page: A User can view and changes campaign details. He can upload images and select users for selection and annotation task. When campaign is ready then a user can start the campaign.
5	Create Running Campaign Details page: A owner of the campaign can view the selected and annotated images.
6	Create Archive Campaign Details page: A owner of the campaign can view the statistics of the terminated campaign.
7	Do the manual changes to the generated code.

Table 5.1: Crowd-Sourcing sprints description

conflicts because both users have changed the same content. Whenever there is a conflict between two files or a conflict is resolved by the user, *Almost-Rerere* intervenes to resolve the conflict or to record the resolved conflict.

The following codes show an example of a template that generates the HTML of a Form View Component, and how it is modified by two developers:

```
1 <h1><%=name %></h1>
2 <form>
3 <% for (var i = 0; i < fields.length; i += 1) { -%>
4     <div class="form-group" data-bind="css: {'has-error': errors
          ()['<%=fields[i]%>']}">
5         <label for="<%=id%>_field_<%=i%>" data-bind="css: {
              active: fields()['<%=fields[i]%>']}, attr: {'data-
```

```
            error': errors()['<%=fields[i]%>']}" class="control-
            label"><%=fields[i]%></label>
6           <input id="<%=id%>_field_<%=i%>" data-bind="value:
            fields()['<%=fields[i]%>']" type="text" class="form-
            control validate" aria-describedby="<%=id%>_field_<%=
            i%>_error">
7           <span id="<%=id%>_field_<%=i%>_error" class="help-block"
             data-bind="text: errors()['<%=fields[i]%>']"></span>
8       </div>
9   <% } -%>
10  </form>
11  <% if (events.length) { -%>
12  <div>
13  <%      for (var i = 0; i < events.length; i += 1) {
14              if (events[i].stereotype == 'system') { -%>
15      <c-<%= events[i].id %> params="trigger: trigger.bind($data
            ,'<%= events[i].id %>')"></c-<%= events[i].id %>>
16  <%          } else { -%>
17      <a class="btn btn-primary" data-bind="click: trigger.bind(
            $data, '<%= events[i].id %>')"><%= events[i].name %></a>
18  <%          }
19          } -%>
20  </div>
21  <% } -%>
```

Both developers make changes to this template on line 2 and 12 in the following way:

```
1  <!-- First Developer -->
2  ...
3  <form class = "red">
4  ...
5  </form>
6  ...
7  <div class = "row">
8  ...
9  </div>
10 ...
```

```
1  <!-- Second Developer -->
2  ...
3  <form class = "blue">
4  ...
5  </form>
6  ...
7  <div class="row">
8  ...
9  </div>
10 ...
```

The changes made concern HTML, JavaScript and CSS templates. In 7 sprints we resolved about 200 conflicts. We will present and discuss all the results in 5.3.1.

5.2 Evaluation of Git project repositories

We needed to test *Almost-Rerere* on a real project to evaluated the quality of the regular expressions and resolutions when there are hundreds of similar conflicts. We used some data sets that are used in [53]. The authors of the cited paper, in order to evaluate their algorithm created data sets based on 9 git-repositories from active open-source projects. They stored and used all commits to the master branch. The data sets concern only Java language, and they contain the pairs of strings: the string in the original state to which we will refer as a *conflict*; the string after the changes to which will refer as a *resolution*.

We have created a program that allows *Almost-Rerere* to iterate over the conflict and resolution pairs directly from our data set. *Almost-Rerere* verifies if it can resolve the conflict by searching for the nearest clusters to the conflict and applying regular expression that was derived for it. The resolution obtained by *Almost-Rerere* is compared with the original resolution (ground-truth) using the Jaro-Winkler similarity score. The conflict, regular expression, replacement, similarity score and resolution is written on a CSV file. Then Almost-Rerere writes the conflict and resolution pair in the nearest cluster.

Repository	**N° Conflicts**
Ant	10500
Cobertura	1260
Eclipse	1355
FitLibrary	4399
JGraphT	3200
JUnit	4424

Table 5.2: Git repository statistics

We use only 6 of 9 repositories used in the cited paper, table 5.2 shows the total number of pairs (conflict & resolution) for each used repository. We will discussed the result in 5.3.2.

Sprint	N° Conflicts	N° Conflicts Resolved
1	2	0
2	11	4
3	11	10
4	24	20
5	40	15
6	71	43
7	49	26
Total	**208**	**118**

Table 5.3: Crowd-Sourcing: Almost-Rerere conflict resolved statistics

5.3 Discussion

In Section 5.3.1 we will present the results of the first case study, Crow-Sourcing. In Section 5.3.2 we will discuss the results of second case study and evaluate the quality of the synthesized resolution by *Almost-Rerere* compared to the original resolution.

5.3.1 Crowd-Sourcing

Table 5.3 shows the total number of conflicts and the number of conflicts resolved by *Almost-Rerere* in 7 sprints. In the first sprint, we have 0 conflicts resolved because there are no recorded conflicts. From the second sprint, *Almost-Rerere* begins to propose a possible solution to the user. In the second sprint, there are only 4 conflicts resolved because there are only a few conflicts in clusters. Then as the number of conflicts increases also the number of resolved conflicts increased.

We have over 200 conflicts, *Almost-Rerere* proposed a resolution to the user for 118 conflicts, that are 57% of total conflicts. The quality of the resolution depends on the intra-cluster similarity of the cluster, which defines the similarity of elements in a cluster.

Table 5.4 shows the total number of clusters, and for each cluster total number of conflicts and of regular expressions & replacements. *Almost-Rerere* creates 21 clusters and records in these clusters 121 different conflicts. We have 7 clusters with intra-cluster similarity above 90%, some of them have a few numbers of conflicts, but all of them give a good conflict resolution. In the other 7 cases, we have clusters with intra-cluster similarity less than 90%, in these cases we will not have a

good resolution in every case. It is necessary to increase the number of conflicts in the cluster to make regular expressions more precise. In the remaining cases, nothing can be said because there is only one conflict per cluster.

Cluster	N° Conflicts	Intra-cluster similarity %
1	4	99
2	3	93
3	10	90
4	1	0
5	2	86
6	6	87
7	2	85
8	10	91
9	26	83
10	2	83
11	1	0
12	1	0
13	1	0
14	1	0
15	1	0
16	1	0
17	2	92
18	17	94
19	11	83
20	15	93
21	4	85

Table 5.4: Crowd-Sourcing: Almost-Rerere cluster statistics

Initially, the proposed solution is very sensible because there are few numbers of records conflicts in clusters. If a cluster has an intra-cluster similarity score above 90% then the resolution obtained after applying a regular expression & replacement is very good. For example, the following cluster have almost the same conflicts, they have little difference like spaces.

```
1 "1":[
2     {
3       "conflict":"<form class=\"blue\">",
4       "resolution":"<form class=\"red\">"
```

```
    },
    {
      "conflict":"<form class = \"blue\">",
      "resolution":"<form class =\"red\">"
    },
    {
      "conflict":"<form class = \"blue\">",
      "resolution":"<form class =\"red\" >"
    },
    {
      "conflict":"<form class = \"blue\">",
      "resolution":"<form class = \"red\" >"
    }
  ],
```

This cluster has following regular expressions & replacements:

```
"1": [
        {
            "regex": "[^7]\"[^;]\\w*+",
            "replacement": "\"red"
        },
        {
            "regex": "(?:([^2])[^_]u(e)(\"))++",
            "replacement": "red$3 "
        }
    ],
```

First regular expression is more precise than the second one because the first one can replace every word with the "red" within the quotation marks. For example, if we have the following conflict:

```
<form class = "green">
```

The following regular expression

```
{
    "regex": "[^7]\"[^;]\\w*+",
    "replacement": "\"red"
},
```

applied to the conflict gives the following result, that is similar to our conflict.

```
<form class ="red">
```

But this regular expression is sensitive to the spaces. If we remove a space character in the conflict, as in the following,

```
<form class ="green">
```

the first regular expression this time gives the following resolution. That is similar to original conflict but at compile time it will give an error because there is missing a "=" character.

```
<form class "red">
```

The following second regular expression, instead, is dependent to the character "u" and "e", so it is not able to replace a word within the quotation marks if it will not have any character "u" followed by a character "e".

```
{
    "regex": "(?:([^2])[^_]u(e)(\"))++",
    "replacement": "red$3 "
}
```

For example, if we have the following conflict:

```
<form class = "SkyBlue">
```

it will have the following resolution after the application of second regular expression & replacement.

```
<form class = "Skyred" >
```

But if have the following conflict like the first example, the second regular expression is not able to find the pattern and to apply the replacement.

```
<form class = "green">
```

These examples show that Almost-Rerere can learn a pattern from a set of similar conflicts and then resolve the conflicts. But the learned pattern depends on the number of conflicts in the cluster and how much these conflicts are similar to each other. As we saw *Almost-Rerere* generates two regular expressions & replacements to make them independent of the order of the conflicts. In the first example, the regular expression is more precise than the second example, but in both examples, they are sensitive to small changes because there are only a few conflicts in the cluster.

The regular expressions become more robust and less sensitive to small changes or to order if the number of conflicts in a cluster is large and the conflicts have a high intra-cluster similarity score (above 90%) such as clusters 18 and 20. For example, cluster 20 have following regular expression & replacement.

```
"20": [
        {
            "regex": "(?:(\")\\w++)++",
            "replacement": "$1header"
        },
        {
            "regex": "(\")\\w++(\")",
            "replacement": "$2header$1"
        }
    ],
```

Both regular expressions are very precise to replace the content within the quotation marks, as shown in the following example they give the same result.

```
<!-- Conflict -->
<thead class = "HEAD">

<!-- Resolution First regex & replacement -->
<thead class = "header">

<!-- Resolution Second regex & replacement -->
<thead class = "header">
```

Cluster 9 has the highest number of conflicts but it has an intra-cluster similarity of 83%. In effect, the conflicts in the cluster have the same structure but differ in content. In the following, we show some example of cluster 9 to give the idea of differences.

```
"9":[
        {
            "conflict":"<h3>image<\/h3>",
            "resolution":"<h3>Images<\/h3>"
        },
        {
            "conflict":"<h3>worker<\/h3>",
            "resolution":"<h3>Workers<\/h3>"
        },
        {
            "conflict":"<h3>Annotated image<\/h3>",
            "resolution":"<h3>Annotated Images<\/h3>"
        },
        {
            "conflict":"<h3>worker details<\/h3>",
            "resolution":"<h3>Worker Detail<\/h3>"
        },
        {
            "conflict":"<h1>workers<\/h1>",
```

```
20              "resolution":"<h2>workers<\/h2>"
21          }
22      ],
```

The regular expressions of the this cluster are the following,

```
"9": [
        {
            "regex": "1([^9]++)",
            "replacement": "2$1"
        },
        {
            "regex": "1(>)",
            "replacement": "2$1"
        }
    ],
```

both patterns are dependent on a particular form of the conflict, they replace the conflicts that have a character "1" or "1>". In the following we show two examples.

```
<!-- First Example: conflict -->
<h1>workers<\/h1>

<!-- Resolution: First regex & replacement -->
<h2>workers<\/h1>

<!-- Resolution: Second regex & replacement -->
<h2>workers<\/h2>

<!-- Second Example: conflict -->
<h3>Annotated image<\/h3>
```

Second resolution is better than the first one because it replaces "h1" with "h2". But in the second example, both the regular expressions are not able to find a pattern to replace.

To make regular expressions independent of a particular character, the similarity of the cluster must be very high. Furthermore, to make regular expressions independent of order and insensitive to small changes it is necessary to have a high number of conflicts in the cluster.

5.3.2 Git project repositories

Almost-Rerere has found a similar resolution for a certain number of conflicts for each repository: Ant 54%, Cobertura 70%, Eclipse 59%, FitLibrary 54%, JGrapT 68%, JUnit 49%. Table 5.5 shows statistics

of the repositories that we evaluate with *Almost-Rerere*. *Almost-Rerere* resolved 55,7% of conflicts.

Repository	**N° Conflicts**	**N° Cluster**	**N° Conflicts Resovled**
Ant	10500	1294	5667
Cobertura	1260	179	885
Eclipse	1355	382	799
FitLibrary	4399	337	2371
JGraphT	3200	238	2135
JUnit	4424	388	2166
Total	**25138**	**2818**	**14023**

Table 5.5: Repositories: Almost-Rerere cluster and conflicts resolved statistics

To evaluate the quality of synthesized resolutions by *Almost-Rerere* we compared with the original resolution using Jaro-Winkler similarity score. Table 5.6 shows the number of conflicts for similarity intervals: 100 - 90 %, synthesized resolution is almost similar to the original; 89 - 80 %, synthesized resolution is similar in some parts to the original, 79 - 0 %, synthesized resolution is not similar to the original. *Almost-Rerere* has resolved 14023 conflicts, 65,8% of resolutions have a similarity score, compared to the original resolution, over 90%. So, the synthesized resolution by *Almost-Rerere*, based on previous similar conflicts, is almost similar to the original resolution in 65,8% of cases.

Repository	**100 - 90 %**	**89 - 80 %**	**79 - 0 %**
Ant	3717	791	1159
Cobertura	577	100	208
Eclipse	567	128	104
FitLibrary	1690	434	247
JGraphT	1748	221	166
JUnit	1470	370	326
Total	**9229**	**2044**	**2210**

Table 5.6: Repositories: Almost-Rerere N° conflicts for similarity intervals statistics

Table 5.7 shows number of clusters of each intra-cluster similarity interval. Our threshold for Jaro-Winkler similarity score is 80%, as explained in 4.1.1, below that all the clusters have only one component and intra-cluster similarity score is equal to 0 because they are not similar to previous conflicts.

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

Table 5.7: Repositories: Almost-Rerere N° of cluster for intra-cluster similarity intervals statistics

Now we will show some interesting case of clusters where *Almost-Rerere* is able to synthesize a resolution with high similarity score even if there are conflicts that have the same structure but different content.

Repository Ant has a cluster with the intra-cluster similarity of 83% and it has 296 pairs of conflict and resolution. *Almost-Rerere* synthesize a resolution of 276 conflicts out of 296. In the following we show some examples of conflicts presents in the cluster:

```
"13": [
        {
            "conflict": "public void setZipfile(File zipFile)
                {",
            "resolution": "public void setZipfile(final File
                zipFile) {"
        },
        {
            "conflict": "public void setCommand(String command)
                {",
            "resolution": "public void setCommand(final String
                command) {"
        },
        {
            "conflict": "public void setComment(String comment)
                {",
            "resolution": "public void setComment(final String
                comment) {"
        },
        {
            "conflict": "public void setVerbose(boolean b) {",
            "resolution": "public void setVerbose(final boolean
                b) {"
        },
        {
```

```
19              "conflict": "public void setProxy(Object o);",
20              "resolution": "void setProxy(Object o);"
21          }
22      ]
```

This cluster has the following regular expression:

```
"13": [
        {
            "regex": "(\\()(b)",
            "replacement": "$1final $2"
        },
        {
            "regex": "(\\()(\\w)",
            "replacement": "$1final $2"
        }
    ],
```

Both regular expression gives a good resolution. But the First one is dependent on a character "b", instead the second one is the more general, it can always find a pattern in a conflicts. if we have the following conflict, the first regular expression & replacement do not give any result but the second gives a good resolution that is almost to the original resolution.

```
<!-- conflict -->
public void log(String message) {

<!-- First regular expression & replacement -->
No resolution

<!-- Second regular expression & replacement -->
public void log(final String message) {
```

There are a few conflicts in the cluster for which the first regular expression do not give any resolution and the second gives a resolution that is different from the original resolution, as shown in the following.

```
<!-- conflict -->
public void setProxy(Object o);

<!-- Original resolution -->
void setProxy(Object o);

<!-- First regular expression & replacement -->
No resolution

<!-- Second regular expression & replacement -->
void setProxy(final Object o);
```

Repository Cobertura has a cluster which is composed of 68 conflicts and has an intra-cluster similarity score of 96%. In the following we show some examples of conflict in the cluster:

```
"29": [
        {
            "conflict": "final private boolean jj_3R_84() {",
            "resolution": "private boolean jj_3R_87() {"
        },
        {
            "conflict": "final private boolean jj_2_1(int xla)
                {",
            "resolution": "private boolean jj_2_1(int xla) {"
        },
        {
            "conflict": "final private boolean jj_2_2(int xla)
                {",
            "resolution": "private boolean jj_2_2(int xla) {"
        },
        {
            "conflict": "final private boolean jj_2_22(int xla)
                {",
            "resolution": "private boolean jj_2_22(int xla) {"
        },
    ]
```

Initially, when there were few conflicts *Almost-Rerere* generated a regular expression & replacement that gave very bad resolution. Around 50 conflicts, regular expressions became more precise and started to give a good resolution. After 68 conflicts, the synthesized resolution is almost similar to the conflict, in fact, *Almost-Rerere* learned that the real difference between conflict and resolution was the word "final". In the following we show an example.

```
"29": [
        {
            "regex": "[^r]++",
            "replacement": "p"
        },
        {
            "regex": "[^p]++\\w",
            "replacement": "p"
        }
    ],
```

```
<!-- conflict -->
```

```
2  final private boolean jj_2_29(int xla) {
3
4  <!-- Original resolution -->
5  private boolean jj_2_29(int xla) {
6
7  <!-- First regular expression & replacement -->
8  prp
9
10 <!-- Second regular expression & replacement -->
11 private boolean jj_2_29(int xla) {
```

The first regular expression & replacement gives a very bad resolution. Instead, the second one gives resolution similar to the original resolution. This example demonstrates that high intra-cluster similarity does not always mean a good resolution. It is important to have a high number of similar pairs of conflicts and resolutions to have always a good resolution.

JGraphT repository has a cluster with 40 conflicts and intra-cluster similarity of 84%. Here are some example of conflicts:

```
1  "81": [
2          {
3              "conflict": "public ConnectivityInspector(
                   UndirectedGraph g ) {",
4              "resolution": "public ConnectivityInspector(
                   UndirectedGraph<V, E> g ) {"
5          },
6          {
7              "conflict": "public BreadthFirstIterator( Graph g )
                   {",
8              "resolution": "public BreadthFirstIterator( Graph<V,
                    E> g ) {"
9          },
10         {
11             "conflict": "public CycleDetector( DirectedGraph
                   graph ) {",
12             "resolution": "public CycleDetector( DirectedGraph<V
                   , E> graph ) {"
13         },
14         {
15             "conflict": "public TopologicalOrderIterator(
                   DirectedGraph dg ) {",
16             "resolution": "public TopologicalOrderIterator(
                   DirectedGraph<V, E> dg ) {"
17         },
18     ]
```

This is a very interesting example because *Almost-Rerere* converges to a single regular expression which is the following.

```
"81": [
        {
            "regex": "(h)( )",
            "replacement": "$1<V,$2E> "
        }
    ]
```

If we have following conflicts, the regular expression gives a resolution that is equal to the original resolution.

```
<!-- Conflitc -->
public BreadthFirstIterator( Graph g ) {

<!-- Original resolution -->
public BreadthFirstIterator( Graph<V, E> g ) {

<!-- Regex & expression resolution -->
public BreadthFirstIterator( Graph<V, E> g ) {

<!-- Conflitc -->
public ConnectivityInspector( DirectedGraph g ) {

<!-- Original resolution -->
public ConnectivityInspector( DirectedGraph<V, E> g ) {

<!-- Regex & expression resolution -->
public ConnectivityInspector( DirectedGraph<V, E> g ) {
```

In the first example, we showed *Almost-Rerere* can synthesize a good resolution even if there are few conflicts in the cluster that have the same structure and similar content. If, instead, the conflicts have the same structure but different content, *Almost-Rerere* needs more example to synthesize a good resolution that we demonstrated in the second example.

Chapter 6

Conclusions and Future Work

Almost-Rerere is an approach aimed to resolve automatically similar conflicts to speed up development times.

In Crowd-Sourcing example we analyzed different examples of resolution synthesized by *Almost-Rerere*. The synthesis of the regular expression by *Almost-Rerere* become more precise and more robust if the number of conflicts and intra-cluster similarity is high. *Almost-Rerere* proposed a solution for 57% of conflicts and generated 66% of clusters with intra-cluster similarity over 80%.

In the second example, we evaluated the quality of resolution synthesized by *Almost-Rerere*, comparing it to the original resolution 66% of synthesized resolution have a similarity score over 90%, which is a very good result. Thus a developer could accept the solution proposed by *Almost-Rerere* in 66% of cases, reducing the number of conflicts.

The future work will focus on the following aspects:

1. The algorithm for search and replace of the regular expression tries to generalize a common regular expression & replacement to cover changing section of multiple samples. This decreases the accuracy of the regular expressions as the samples similarity decreases. The algorithm can be improved by considering several changing regions on the same sample, instead of considering the changing region as a whole. Suppose we have two following strings:

```
1  Original: <h1>Thesis<h1>
2  Modified: <h2>Thesis<h2>
```

   Algorithm compares two strings character by character from start and it stops when finds a different character. Then do the same

starting by end of the string. It separates strings in three areas: $1 and $3 are the same in both strings, and $2 is the difference. Algorithm will generate a regular expression considering the whole area $2 as a changing region.

```
1    <h | 1 >Thesis<h 1 | >
2    <h | 2 >Thesis<h 2 | >
3    $1 |       $2        | $3
```

We want that algorithm separates $2 in more regions as shown in following, and generates regular expressions only for $2 and $4 areas.

```
1    <h | 1   | >Thesis<h | 1   | >
2    <h | 2   | >Thesis<h | 2   | >
3    $1 | $2 |     $3      | $4 | $5
```

2. This version of *Almost-Rerere* can handle only single-line conflict because it is very likely to have single-line conflicts similar to the previous one respect to multi-line conflicts. Multi-line conflicts means that the conflict area has two or more consecutive lines that differ. In future work, we want to improve our approach to deal with this case as well.

3. Improve the clustering algorithm to make clusters with high intra-cluster similarity. We want to improve the threshold for Jaro-Winkler similarity to have clusters with high intra-cluster similarity. As a result, the regular expressions & replacements will also be improved.

4. Implement an algorithm to re-cluster the conflicts to make the cluster independent of the initial conflicts.

5. Create a graphical interface for *Almost-Rerere* that allows the developers to select the proposed resolution.

Bibliography

[1] Carlo Bernaschina. Almost. js: an agile model to model and model to text transformation framework. In *International Conference on Web Engineering*, pages 79–97. Springer, 2017.

[2] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006.

[3] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[4] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[5] Marco Brambilla and Piero Fraternali. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014.

[6] S. Kelly and J.P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley - IEEE. Wiley, 2008.

[7] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52:43 – 62, 2018.

[8] Geoffrey Sparks. Enterprise architect user guide, 2009.

[9] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93. ACM, 2003.

[10] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. *Software Language Engineering, SLE*, 16(3), 2010.

[11] Jetbrains, meta programming system,http://jetbrains.com/mps.

[12] Markus Voelter. Projectional language workbenches as a foundation for product line engineering. *Software Engineering 2010–Workshopband (inkl. Doktorandensymposium)*, 2010.

[13] Richard C Gronback. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Educzation, 2009.

[14] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[15] Carlo Bernaschina, Sara Comai, and Piero Fraternali. Ifmledit. org: model driven rapid prototyping of mobile apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 207–208. IEEE Press, 2017.

[16] Carlo Bernaschina, Emanuele Falzone, Piero Fraternali, and Sergio Luis Herrera Gonzalez. The virtual developer: Integrating code generation and manual development with conflict resolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):20, 2019.

[17] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, et al. A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 74–85. IEEE, 2015.

[18] Eclipse. https://www.eclipse.org/acceleo/, 2007.

[19] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering*, 43(5):396–414, 2016.

[20] Jokin García, Oscar Diaz, and Maider Azanza. Model transformation co-evolution: A semi-automatic approach. In *International Conference on Software Language Engineering*, pages 144–163. Springer, 2012.

[21] Larry W Allen, Gary L Fernandez, Kenneth P Kane, David B Leblang, Debra A Minard, and Gordon D McLean Jr. Version control system for geographically distributed software development, October 7 1997. US Patent 5,675,802.

[22] Filippo Lanubile. Collaboration in distributed software development. In *Software Engineering*, pages 174–193. Springer, 2007.

[23] Git. https://git-scm.com/.

[24] Mercurial. https://www.mercurial-scm.org/.

[25] Apache Subversion. https://subversion.apache.org/.

[26] Eoin Ó Conchúir, Pär J Ågerfalk, Helena H Olsson, and Brian Fitzgerald. Global software development: where are the benefits? *Communications of the ACM*, 52(8):127–131, 2009.

[27] Linus Trovald. https://www.kernel.org/.

[28] https://backlog.com/blog/git-vs-svn-version-control-system/.

[29] Tom Mens. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462, 2002.

[30] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.

[31] Antonio Cicchetti, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Managing the evolution of data-intensive web applications by model-driven techniques. *Software & Systems Modeling*, 12(1):53–83, 2013.

[32] Mendix. https://www.mendix.com.

[33] WebRatio. https://www.webratio.com.

[34] Outsystems. https://www.outsystems.com.

[35] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 21–30, Honolulu, Hawaii, October 2008. Association for Computational Linguistics.

[36] Falk Brauer, Robert Rieger, Adrian Mocan, and Wojciech M. Barczynski. Enabling information extraction by inference of regular expressions from sample entities. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 1285–1294, New York, NY, USA, 2011. ACM.

[37] Cherie Madarash-Hill and JB Hill. Enhancing access to ieee conference proceedings: a case study in the application of ieee xplore full text and table of contents enhancements. *Science & Technology Libraries*, 24(3-4):389–399, 2004.

[38] Efim Kinber. Learning regular expressions from representative examples and membership queries. In *International Colloquium on Grammatical Inference*, pages 94–108. Springer, 2010.

[39] David F Barrero, David Camacho, and Maria D R-moreno. Automatic web data extraction based on genetic algorithms and regular expressions. In *Data Mining and Multi-agent Integration*, pages 143–154. Springer, 2009.

[40] Antonio González-Pardo, David F Barrero, David Camacho, and María D R-Moreno. A case study on grammatical-based representation for regular expression evolution. In *Trends in Practical Applications of Agents and Multiagent Systems*, pages 379–386. Springer, 2010.

[41] Bertrand Daniel Dunay, Frederick E Petry, and Bill P Buckles. Regular language induction with genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 396–400. IEEE, 1994.

[42] Borge Svingen. Learning regular languages using genetic programming. *Proc. Genetic Programming*, pages 374–376, 1998.

[43] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Marco Mauri, Eric Medvet, and Enrico Sorio. Automatic generation of regular expressions from examples with genetic programming. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 1477–1478. ACM, 2012.

[44] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Learning text patterns using separate-and-conquer genetic programming. In *European Conference on Genetic Programming*, pages 16–27. Springer, 2015.

[45] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Active learning of regular expressions for entity extraction. *IEEE transactions on cybernetics*, 48(3):1067–1080, 2017.

[46] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Automatic search-and-replace from examples with coevolutionary genetic programming. *IEEE transactions on cybernetics*, 2019.

[47] Automatic Generation of Text Extraction Patterns from Examples. http://regex.inginf.units.it/.

[48] Almost-Git. https://www.npmjs.com/package/almost-git.

[49] Fionn Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[50] Jan Martin Keil. Efficient bounded jaro-winkler similarity based search. *BTW 2019*, 2019.

[51] William Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. *Proceedings of the Section on Survey Research Methods*, 01 1990.

[52] https://editor.ifmledit.org/.

[53] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M Eskofier, and Michael Philippsen. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 61–72. ACM, 2016.