CoRA: Decomposing and Describing Tangled Code Changes for Reviewer

Min Wang*
Peking University
Beijing, China
wangmin1994@pku.edu.cn

Zeqi Lin
Microsoft Research
Beijing, China
Zeqi.Lin@microsoft.com

Yanzhen Zou*†
Peking University
Beijing, China
zouyz@pku.edu.cn

Bing Xie*
Peking University
Beijing, China
xiebing@pku.edu.cn

***Abstract*—Code review is an important mechanism for code quality assurance both in open source software and industrial software. Reviewers usually suffer from numerous, tangled and loosely related code changes that are bundled in a single commit, which makes code review very difficult. In this paper, we propose *CoRA* (Code Review Assistant), an automatic approach to decompose a commit into different parts and generate concise descriptions for reviewers. More specifically, *CoRA* can decompose a commit into independent parts (e.g., bug fixing, new feature adding, or refactoring) by code dependency analysis and tree-based similar-code detection, then identify the most important code changes in each part based on the PageRank algorithm and heuristic rules. As a result, *CoRA* can generate a concise description for each part of the commit. We evaluate our approach in seven open source software projects and 50 code commits. The results indicate that *CoRA* can improve the accuracy of decomposing code changes by 6.3% over the state-of-art practice. At the same time, *CoRA* can identify the important part from the fine-grained code changes with a mean average precision (MAP) of 87.7%. We also conduct a human study with eight participants to evaluate the performance and usefulness of *CoRA*, the user feedback indicates that *CoRA* can effectively help reviewers.**

***Index Terms*—Code review, Code changes decomposition, Code changes description, Program comprehension**

I. Introduction

Code review is a common and important software engineering practice during software development, which is recognized as a valuable way for preventing software defects and improving code quality in software projects [15] [16] [17] [18] [19] [20] [38] [45]. Software undergoes continuous changes, through which new features are added, bugs are fixed, and performance is improved. These code changes usually need to be understood by software engineers when performing their daily development and maintenance tasks, in particular during the code review process [1] [21] [39] [46].

However, developers often bundle unrelated or loosely related code changes (e.g., bug fixing and refactoring) in a single commit, thus resulting in so-called tangled commit [8]. In Herzig and Zeller's study [8], they confirmed that tangled code changes occur frequently, with up to 15% of bug fixing change sets applied to the subject projects being tangled. And as Tao et al. [1] pointed out in an exploratory study in industry, understanding tangled code changes requires non-trivial efforts [47], and a tool feature for change decomposition is desired.

To address the problem of tangled code changes, researchers proposed various tools for decomposing code changes in a commit [8] [9] [11] [12]. Most of their work considered lightweight code dependencies (e.g., *def-use* information) for clustering code changes. However, during the code review process, reviewers prefer to get a complete view of the commit, which requires the tool to be able to analyze more comprehensive relationships between code changes (e.g., similarity between code changes, method-override, and class-extends), rather than just considering some lightweight code dependencies [18] [48]. Moreover, there are still many trivial fine-grained code changes in a cluster after the decomposition. Most of the previous work does not further analyze and deal with these trivial code changes (e.g., estimating the importance of fine-grained code changes in a cluster), which makes the code review still difficult [49].

Our goal is to decompose and describe tangled code changes in a commit so as to provide a compact overview of the commit to reviewers. Compared with previous work, our insights are: (1) we provide a richer set of relations between fine-grained code changes, so that each commit can be decomposed into several clusters; (2) we generate a concise natural language description for each cluster, which is helpful for reviewers to understand the commit.

The main challenges we face are:

- To decompose tangled code changes better, we need to propose a more accurate code changes decomposition solution by deep mining more complex and comprehensive relationships between programs (e.g., similarity between code changes [50] [51]).
- To generate concise descriptions for each code change cluster, we need to identify the importance of fine-grained code changes in each cluster. As a cluster may involve many trivial code changes [13], it will obviously be very verbose and impractical if we describe all these fine-grained changes without primary and secondary points.

In this paper, we propose *CoRA* (**C**ode **R**eview **A**ssistant), an automatic approach for decomposing and describing tangled code changes. Firstly, *CoRA* decomposes a tangled commit into several code change clusters. Fine-grained code

* Also with the Key Laboratory of High Condence Software Technologies, Ministry of Education, Beijing, China and the School of Electronics Engineering and Computer Science, Peking University, Beijing, China.
† Y. Zou is the Corresponding author.

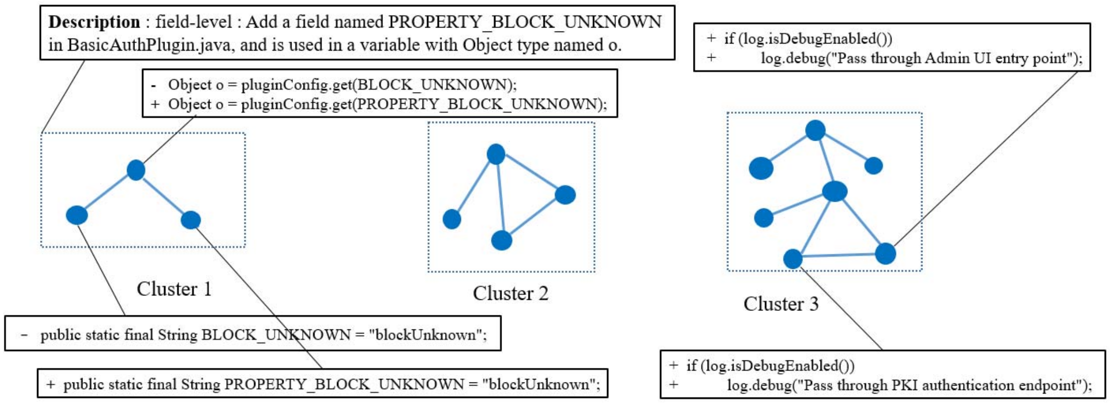
Description : field-level : Add a field named PROPERTY_BLOCK_UNKNOWN in BasicAuthPlugin.java, and is used in a variable with Object type named o.
- Object o = pluginConfig.get(BLOCK_UNKNOWN);
+ Object o = pluginConfig.get(PROPERTY_BLOCK_UNKNOWN);
+ if (log.isDebugEnabled())
+ log.debug("Pass through Admin UI entry point");
Cluster 1
Cluster 2
Cluster 3
- public static final String BLOCK_UNKNOWN = "blockUnknown";
+ public static final String PROPERTY_BLOCK_UNKNOWN = "blockUnknown";
+ if (log.isDebugEnabled())
+ log.debug("Pass through PKI authentication endpoint");

Fig. 1. An Example of Code Changes from Commit 280f679 in lucene-solr

changes are clustered based on dependency and similarity relationships between them. This is inspired by and extended from ClusterChanges [9], which clusters code changes based on dependency relationships. Compared with ClusterChanges, *CoRA* has extra consideration of the similarity between code changes (i.e., tree similarity [2] between two abstract syntax trees), which clusters implicitly related code changes together. Then, *CoRA* generates natural language descriptions for each code change cluster respectively: the importance of each fine-grained code change in a cluster is estimated mainly based on the PageRank algorithm [22], so that *CoRA* can describe these code changes with primary and secondary points; natural language descriptions are generated based on the combination of natural language templates and program analysis techniques, which help code reviewers understand each cluster.

We conducted experiments on seven open source Java projects to compare the accuracy of *CoRA*'s tangled commit decomposition module against the state-of-art techniques (i.e., ClusterChanges [9]). Moreover, we conducted an objective human study with eight participants to evaluate the usefulness and effectiveness of *CoRA*. In a comprehensive evaluation, the results demonstrate that (1) *CoRA* is effective at tangled commit decomposition (in the seven open source projects, it achieved a median accuracy of 90%, which is better than the state-of-art techniques); (2) most of the natural language descriptions for code change clusters are reasonable for providing a compact overview of a commit, in particular, for identifying the most important parts in each cluster; (3) descriptions generated by *CoRA* are useful for reviewers in understanding code changes during the code review process.

In summary, this work makes the following contributions.

- We propose a novel method to decomposing a commit into several clusters, in which not only code change dependencies but also code change similarities are incorporated to cluster fine-grained code changes.
- We propose a novel method to generating natural language descriptions for each code change cluster, in which the importance of each fine-grained code change is estimated so that we can prioritize descriptions for more important code changes.
- We propose *CoRA*, an approach and corresponding tool to automatically decompose and describe tangled commits for reviewers. Experiments conducted on seven open source projects show that *CoRA* is effective to help code reviewers understand tangled commits.

II. Motivating Example

In this section, we present a simple tangled commit selected from Apache lucene-solr[1] to further motivate our research.

Fig. 1 presents an example of code changes decomposition result for commit 280f679 of Apache lucene-solr, which consists of three clusters. In this commit, there are 92 lines of code changes involving three Java class files with 64 additions and 28 deletions. We manually untangled this commit into three independent clusters, and extracted some fine-grained code changes in the *diff* file. In cluster 1, the declaration of a *field* named *PROPERTY_BLOCK_UNKNOWN* is added, the declaration of a *field* named *BLOCK_UNKNOWN* is deleted, and thus the *variable* named *o* is updated. We found these code changes in cluster 1 can be understood as a completely separate part. Similarly, in cluster 2, there are also code changes caused by definition and usage. Besides the *def-use* information, we can discover more dependency relationships between code changes. For example, if an abstract method has been changed, thus the corresponding override method will also be changed. Similarly, there are also some program dependencies (e.g., class extends other class) between class and class. Even though the *def-use* relationship is the most common in program, we believe that mining more program dependencies will make the code changes decomposition more accurate, especially when the code changes are very complicated. So it motivates us to mine more comprehensive and complex program dependencies between code changes.

A more common occurrence in code changes is to modify similar code for the same purpose, and it is worthwhile to mine this similarity relationship between code changes. For

[1] https://github.com/apache/lucene-solr

Table I. Definitions of Different Level Code Changes Based on AST Node Type Hierarchy

DEFINITION	AST Node Type	Description
Class-level Changes	*ClassDeclaration InnerClassDeclaration Interface Enum ...*	a *class-level* code change entity represents a change in the declaration of a class (such as adding a new class or deleting a class). It may cover *method-level* or *field-level* changes.
Method-level Changes	*MethodDeclaration*	a *method-level* code change entity represents a change in the declaration of a method. It may cover *statement-level* changes.
Field-level Changes	*FieldDeclaration*	a *field-level* code change entity represents a change in the declaration of a field.
Statement-level Changes	*SimpleName If For Try ExpressionStatement ...*	a *statement-level* code change entity represents other changes, such as updating a *if* structure or adding a expression.

example, in cluster 3, the core of the code changes is a newly-added *if* structure, and there is another newly-added *if* structure. We can see that there is no any code dependency relationship between these two code snippets, but we found that these two code snippets should be clustered together for reviewers, because it seems that those code snippets are both for the purpose of printing debug log. It will be helpful for code review if we can cluster these similar code changes (i.e., code changes for the same purpose).

Consider the commit message in this commit, *"SOLR-7896: Add a login page to Admin UI, with initial support for Basic Auth"*, as we can see, this commit message reveals the main intention (i.e., *why* did this code change) of the commit from programmer, which is helpful for code review. But actually, in addition to the main intent information, the main code change information (i.e., *what* did in this code change) and internal code logic information is also important for reviewers, especially the commit is tangled. However, we cannot describe all the changes in a commit, so we should identify the most important code changes. For example, in cluster 1, we found that the declaration of the newly-added *field* is more important and it is a field-level code change, so we can describe this cluster just like: " *field-level : Add a field named PROPERTY_BLOCK_UNKNOWN in BasicAuth-Plugin.java, and is used in a variable with Object type named o.*". We believe these descriptions will be helpful for reviewers. Firstly, it provides reviewers with key information on cohesive code changes. Second, it assists reviewers to review those code changes. So it motivates us to generate descriptions by identifying the most important parts in a commit.

To sum up, our idea is to first decompose the code changes into independent parts, which makes a cluster view of the decomposed code changes. For each part, we generate descriptions according to the importance of the fine-grained code changes. So we can provide a compact overview of the commit for code reviewers.

III. Approach

Fig. 2 shows the workflow of our approach. The framework consists of two phases, namely the tangled commit decomposition phase and decomposed clusters description phase.

Tangled Commit Decomposition. This phase aims to decompose the tangled code changes by extracting a code change graph from a commit. In detail, firstly, *CoRA* extracts the commit from the version control system, and converts the previous program and current program into the corresponding Abstract Syntax Tree (AST) respectively. Then, *CoRA* extracts the fine-grained code changes by comparing the differences between the previous ASTs and current ASTs. Next, *CoRA* identifies the dependencies (such as inheritances, declarations and method invocations) between these fine-grained code changes utilizing static program analysis technology. And *CoRA* further explores the similarity between these code changes to cluster the fine-grained changes. Finally, we get a code change graph, each node in the graph stands for a hierarchical fine-grained code changes, and the edges in the graph stand for the relationships between these code changes.

Decomposed Clusters Description. This phase aims to identify the most important parts of code changes so as to describe each cluster. In general, a complete and cohesive code change cluster usually contains a lot of trivial changes, which are negligible for understanding code changes. However, we are unable to obtain the insight about which part of code changes is more important directly from the commit messages. To address it, we are inspired by the PageRank algorithm [22]. *CoRA* obtains directed graphs corresponding each cluster in the above phase. On this basis, *CoRA* utilizes a series of heuristic rules to assign initial value to each node (i.e., fine-grained code changes) in the graph. Then the PageRank algorithm is applied to identify the importance of these fine-grained code changes by calculating the value of each node iteratively. Finally, *CoRA* starts with the most important node and combines the predefined natural language templates to describe each cluster.

A. Decomposing Tangled Commit

In this subsection, we describe the details for decomposing tangled commit.

1) Extracting Fine-grained Code Changes: We extract the change sets between two adjacent versions of a Java project by using the JGit[2]. Then we utilize the GumTree [23] (i.e., a tree-based code changes differencing algorithm) to extract fine-grained code changes by parsing ASTs of two versions of the code. Finally, we obtain the fine-grained code change fragments, i.e., independent lines of code. All fine-grained

[2] https://www.eclipse.org/jgit/

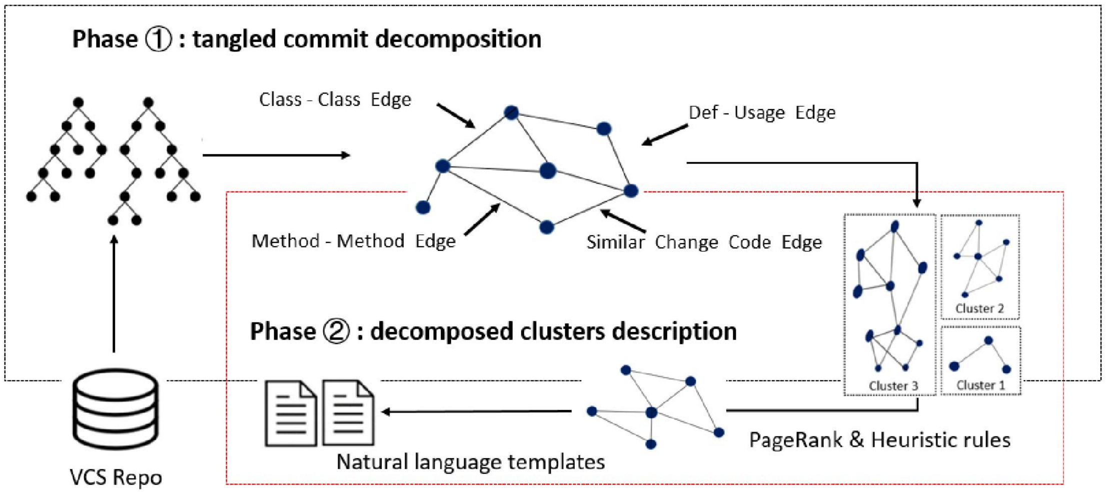
Phase ①: tangled commit decomposition
Class - Class Edge
Def - Usage Edge
Method - Method Edge
Similar Change Code Edge
Cluster 2
Cluster 3
Cluster 1
Phase ②: decomposed clusters description
VCS Repo
Natural language templates
PageRank & Heuristic rules

Fig. 2. Workflow of *CoRA*

code changes can be represented by a triple of (⟨operation type⟩, ⟨AST node type⟩, ⟨code changed content⟩).

2) Generating a code change graph: By parsing and analyzing the AST node type hierarchy, we define these code change fragments as four different levels of code change entities: *class, method, field, statement*. The definition information is shown in Table I. For example, as shown in Fig. 1, the newly-added *field* named *PROPERTY_BLOCK_UNKNOWN* in cluster 1 represents a *field-level* code change entity, and the newly-added *if* structure in cluster 3 represents a *statement-level* code change entity.

We are inspired by ClusterChanges, which clustered code changes based on a dependency graph. We first follow this idea to establish a more comprehensive code dependencies between code change entities. Then we further cluster these fine-grained code change entities by additional process which we will introduce in the next step. We utilize static program analysis technique to establish some code dependencies as follows:

- *Definition-Usage edges:* We first establish the edge between a definition and an usage, i.e., if the declaration of a variable, field or method is changed, we scan all the code change entities to discover the set of usages (i.e., references to this declaration), then we create a directed edge from the usage point to the definition. The *Definition-Usage edges* are the most frequent relationships in code changes, and they are also important criteria for judging whether the code changes are the same cluster.
- *Method-Method edges:* Consider the code dependencies between *method-level* code change entities, we can discover three types of code dependencies: *Abstract-Method, Override-Method, Implement-Method*. In detail, we traverse all *method-level* code change entities to discover *abstract methods, interface methods or methods*, then we can create a directed edge between two *method-level* code change entities if their method signature are same.
- *Class-Class edges:* In code changes, an infrequent but important situation is the change of an abstract class (i.e., add, delete or update an abstract class file), and thus the corresponding inherited class has to be changed. Similarly, we create directed edges between these *class-level* code change entities by traversing all *class-level* code change entities.

The result is a directed graph, which nodes represent these hierarchical code change entities and edges represent the identified code dependencies between these code change entities. We think those code dependencies can reveal those code changes that achieve a joint purpose, and are thus more likely to be part of the same cohesive changes in a tangled commit.

3) Clustering Similar Code Changes: We obtain the initial clusters of a commit by constructing a code change graph. However, just considering the program dependencies between code change fragments is not enough for clustering. A very common situation in code changes is that the developers modify the similar code snippets for closely related purpose (e.g., refactoring, bug fixing), and there is no program dependency between these changed code snippets, but they should belong to the same cluster.

As shown in Fig. 1, cluster 3 presents a simple example of similar code change fragments. The code changes of the first *if* structure and the code changes of the second *if* structure are very similar, and both for the purpose of printing debug log. If we just consider the program dependency, we will miss a lot of code change snippets that should be clustered. To address it, we utilize the APTED (All Path Tree Edit Distance) algorithm [2], which is a state-of-art approach to calculate the similarity of two code change fragments.

In detail, we first convert those code change fragments (i.e., all same types of nodes in the code change graph) into corresponding partial Abstract Syntax Trees (ASTs), we translate the similarity calculation of the code change fragments pair into the similarity calculation of the ASTs pair. Then we adopt the tree edit distance as the standard measure for tree similarity, which is defined as the minimum-cost sequence of node edit operations that transform one tree into another. In particular, we consider following three edit operations on those ASTs:

- **Delete.** It denotes a node deletion from the current AST and connect its children to its parent.
- **Insert.** It denotes a node insertion between an existing node and its children.
- **Update.** It denotes an update in the code content of a node.

Finally, we calculate the value of similarity between two ASTs pair by using APTED algorithm, and then we set a similarity threshold to determine whether the two code change entities should belong to the same cluster. It is well-tested that 0.8 is an appropriate value in our scenario, so we apply this heuristic threshold [2]. If they belong to the same cluster, we create an undirected and unweighted edge between the two entities.

Compared to ClusterChanges [9], we not only consider more complex and comprehensive program dependencies, but also further consider the similarity relationships between code changes. We think this will significantly improve the effect of tangled commit decomposition, which will be verified in later experiments.

B. Describing Decomposed Clusters

In this subsection, we describe the details for summarizing decomposed change clusters.

1) Calculating the Importance of Code Changes: This step aims to identify the importance of code change entities in each cluster based on the PageRank algorithm, which is one of the most popular algorithm for web link analysis [25] [26]. Rahman and Roy [14] proposed a novel approach named *CodeRank* to evaluate the importance of terms in source code. In our work, the PageRank algorithm attempts to evaluate the importance of each code change entity through link analysis. It is based on two heuristic assumptions as follows:

- Assumption 1: Important code change entities are generally referenced by more code change entities.

We fully consider dependency relationships between code changes by using a code change graph which is generated in the above. In a code change graph, nodes represent code change entities (i.e., lines of code with different level), directed edges represent some dependency relationships, i.e., $Usage \longrightarrow Definition$, $OverrideMethod \longrightarrow AbstractMethod$, $SubClass \longrightarrow SuperClass$, etc. We calculate the code change entity's importance value, $PR(V_i)$, as follows:

$$PR(V_i) = \frac{1-\psi}{N} + \psi \sum_{V_j \in M_{V_i}} \frac{PR(V_j)}{L(V_j)} \tag{1}$$

Here, $PR(V_i)$ and $PR(V_j)$ denote the importance value of node V_i (i.e., a code change entity) and node V_j (i.e., another code change entity) respectively. N denotes the number of all code change entities in a cluster, M_{V_i} denotes the set of code change entities, which points to V_i (i.e., $V_j \longrightarrow V_i$). $L(V_j)$ denotes the number of outgoing links of V_j.

- Assumption 2: Important code change entities are generally high-level and complex.

We further consider the complexity of the hierarchical code change entity itself based on a series of heuristic rules as follows:

- The importance of code change entities is related to the code level, we set the weights of *class-level, method-level, field-level, statement-level* to 4, 3, 2, 1 respectively.
- The importance of code change entities is related to the number of code lines in the code change snippets.
- The importance of code change entities is related to the number of relationships with other code change entities.

We calculate the weighted average of these heuristic feature rules to evaluate the importance value of the code change entity, $HR(V_i)$, as follows:

$$HR(V_i) = \alpha * Level_{V_i} + \beta * Lines_{V_i} + \gamma * Rels_{V_i} \tag{2}$$

Here, $Level_{V_i}$ denotes the level of code changes we defined in the heuristic rule, $Lines_{V_i}$ denotes the number of lines of code change entity, $Rels_{V_i}$ denotes the number of relationships of node V_i.

Finally, we combine the score by the PageRank algorithm with the score by the heuristic rule algorithm as the final importance score of the code change entity.

$$Score(V_i) = PR(V_i) + HR(V_i) \tag{3}$$

So we can identify the most important part of a cohesive code change cluster by score.

Parameters and Configurations. Equation 1 is adapted from the PageRank algorithm [25] [26]. In this algorithm, ψ is the damping factor, which denotes the probability of randomly choosing a web page in the context of web surfing by a random surfer. It is well-tested that 0.85 is a very generally applicable value for ψ [14], thus we also set ψ to 0.85 in Equation 1. Moreover, the PageRank algorithm needs to iteratively calculate the importance value of each node until convergence, and we apply a heuristic threshold of 0.0001 for convergence checking as applied earlier [14]. In Equation 2, there are three hypher-parameters: α, β and γ. We adopt a simple method to estimate their values: as our evaluation dataset contains seven software projects (see Section IV), we tuned α, β and γ on one of these projects (the *lucene-solr* project), then applied these tuned parameters to the other six projects.

2) Describing the Code Change Cluster: After calculating the importance ranking of code change entities in a cluster, we automatically summarize the code change clusters based on natural language templates.

For each cluster, we take the most important code change entity as the starting point to derive and summarize each cluster. Each description of the cluster is composed of two elements: 1) specific behavioral information of the most important code change entity (e.g., update a method *fun1()*); 2)

Table II. Natural Language Templates for Descriptions of Code Change Cluster

Type	Natural Language Templates (T)
class-level	T: $\langle action \rangle$ a class named $\langle classname \rangle$ in $\langle filename \rangle$, and lead to $\langle$*code change entity*$\rangle$...
method-level	T: $\langle action \rangle$ a method named $\langle methodname \rangle$ in $\langle filename \rangle$, and be invoked in $\langle$*code change entity*$\rangle$...
field-level	T: $\langle action \rangle$ a field named $\langle fieldname \rangle$ in $\langle filename \rangle$, and be used in $\langle$*code change entity*$\rangle$...
statement-level	$\langle action \rangle$ a statement named $\langle stateinfo \rangle$ in $\langle filename \rangle$, and related to $\langle$*code change entity*$\rangle$...

Table III. Natural Language Descriptions for Clusters of Code Changes of Commit 280f679 in lucene-solr

Cluster	Type	Description
cluster 1	*field level*	add a field named *PROPERTY_BLOCK_UNKNOWN* in *BasicAuthPlugin.java*, and is used in a variable with Object type named *o*.
cluster 2	*method level*	add a new method named *isAjaxRequest* , which be called in method named *authenticationFailure* in *BasicAuthPlugin.java*
cluster 3	*field level*	add a field named *PROPERTY_REALM* in *BasicAuthPlugin.java*, and use it in *if* structure in *Sha256AuthenticationProvider.java*.

derived dependency information related to the most important code change entity (e.g., add a method *fun2()*, which call the updated method *fun1()*). These information is summarized by *CoRA* for generating sentences and describing the cluster using the natural language templates listed in Table II. The first column of Table II represents the hierarchy of the most important code change entity, and the second column of Table II represents the natural language templates. $\langle action \rangle$ represents the operation type of the most important code changes (i.e., add, delete, update, etc). $\langle$*code change entity*$\rangle$... represent the code change entity set, which are derived by the most important code change entity. It is entirely possible for a code change entity to derive multiple other code change entities, here, we summarize the top 5 entities by their importance ranking.

For example, we can describe the code changes example in Fig. 1 as TABLE III. The complete description of a tangled commit is a combination of individual description of each cluster.

IV. EVALUATION

To evaluate *CoRA*'s effectiveness and usability, we have conducted a set of quantitative and qualitative experiments. These experiments address the following questions:

RQ1: To what extent is *CoRA* superior to the state-of-art practice on the tangled commit decomposition task ? And why?

The main object of our research is a tangled commit, and we are concerned about the *CoRA*'s performance on the commit decomposition task. To evaluate it, firstly we manually establish a ground truth in seven open source projects. After that, we make some comparisons between *CoRA* and existing approach (i.e., ClusterChanges [9]). And we further explore why *CoRA* is better than the previous approach.

RQ2: How does the *CoRA* perform in identifying the importance of the code changes?

In our approach, we identify the importance ranking of the code changes by the PageRank algorithm and heuristic rules. We are concerned about the effectiveness of our approach.

RQ3: Does the description generated automatically by *CoRA* can cover the main change information of a commit concisely?

We are concerned about the quality of the description generated automatically by *CoRA*, i.e., which is the evaluation of the widely-used key properties: *content adequacy, conciseness, expressiveness* [6] [27], and we will further explain the specific meaning of these evaluation properties later.

Table IV. Statistics of Seven Subject Projects

Project	Domain	LOC	#C	T-C
lucene-solr	search engine system	1150.6K	31534	11
jfreechart	Java 2D chart library	131.8K	3670	5
okhttp	HTTP+HTTP/2 client	53.5K	3657	6
glide	image load&cache library	68.2K	2302	5
spring-framework	spring framework	621K	18082	7
netty	a network framework	263.3K	9192	6
jruby	Ruby on the JVM	253K	46548	10

A. Evaluation Setup

To answer the above research questions, we evaluate our approach in seven well-known open source Java projects. To ensure the objectivity and versatility of the experiment datasets, we selected open source projects implemented in Java language differing in domain, scale, and the number of commits from Github. Table IV presents the statistics about these projects, including project name, domain, lines of source code, the number of commits and the number of selected tangled commits. It can be seen that the total number of selected tangled commits is 50, and the scale is small. However, we selected the most complicated commits from seven different projects. If our tool can work well on these complex commits, then it can be assumed that our tool can work better on those simpler commits. So building larger datasets has no practical meaning for our evaluation.

1) **Establishing Ground Truth :** To best of our knowledge, there is no uniform ground truth available for existing work on code changes decomposition and commit description. So firstly we establish the ground truth for tangled commits.

Commits Selection. Without having a document describing the applied changes, it is very hard to judge whether code changes are tangled or not, at least for a project outsider [8]. To address it, we applied the similar way as Kim et al. [8] to judge whether a commit is tangled or not, i.e., we identified a commit as tangled if the commit message clearly indicates that the applied changes tackle more than one issue report (e.g. "Fix JRUBY-1080 and JRUBY-1247 on trunk.") or the commit message indicates extra work committed along the issue fix.

Table V. Questions and Possible Answers in the Human-subject Study

Property	Question	Possible Answers
Content adequacy	Considering only the content of the description and not the way it is presented, do you think that the description?	1. Is not missing any information 2. Is missing some information but the missing information is not necessary to understand the commit 3. Is missing some very important information that can hinder the understanding of the commit
Conciseness	Considering only the content of the description and not the way it is presented, do you think that the description?	1. Has no unnecessary information 2. Has some unnecessary information 3. Has a lot of unnecessary information
Expressiveness	Considering only the way the description is presented and not its content, do you think that the description?	1. Is easy to read and understand 2. Is somewhat readable and understandable 3. Is hard to read and understand

Then we randomly sampled a total of 50 tangled commits from the seven projects (see column 5 in TABLE IV).

Participants Selection. To establish the ground truth for the above selected commits, we hired eight participants to manually cluster these fine-grained code changes for each commit. In particular, from our school, we hired eight graduate students who had at least 2-years of experience in Java programming. Two of them had 5-years experience; the average experience was 4 years. All the participants are not the authors of this paper. Besides, in order to ensure the quality of the ground truth, we require that participants who are hired be very familiar with these open source projects. So we used two principles of selecting participants to measure familiarity: Participants had to (1) have used these open source projects for programming and/or (2) have worked on branches of these open source projects (for example, forking them on Github).

Ground Truth Establishment. First we extracted fine-grained code changes for each tangled commit using the CLDIFF [24]. Then the eight participants manually clustered these fine-grained code changes for each commit. In particular, they were divided into groups of two participants, and two participants in the same group labeled the same fine-grained code changes for a commit (i.e, identify two fine-grained code changes whether belong to the same cluster). We also invited participants of other groups to make a further judgment if the original participants' judgments are inconsistent. After that, we applied the same way to manually label the most important code changes in each cluster. Finally, we obtained a complete and objective ground truth for our quantitative experiments later.

2) **Quantitative Experiments Metric:** We mainly conduct quantitative experiments for RQ1 and RQ2.

For RQ1, to evaluate how well a clustering result matches the ground truth, we utilize a standard metric (i.e., Rand Index) from community detection [29]. In particular, we obtain a fine-grained code change graph (Section III), which nodes represent these hierarchical code change entities (i.e., lines of code snippets). Considering all possible pairs of nodes ($n(n-1)/2$ pairs for n nodes), there are four kinds of results obtained from the experiments, namely True Positive (TP), False Negative (FN), True Negative (TN), and False Positive(FP). Specifically:

$$TP : Pair(n_1, n_2) \in C_{\mathit{CoRA}} \cap Pair(n_1, n_2) \in C_{g\_truth}$$

$$FN : Pair(n_1, n_2) \notin C_{\mathit{CoRA}} \cap Pair(n_1, n_2) \in C_{g\_truth}$$

$$TN : Pair(n_1, n_2) \notin C_{\mathit{CoRA}} \cap Pair(n_1, n_2) \notin C_{g\_truth}$$

$$FP : Pair(n_1, n_2) \in C_{\mathit{CoRA}} \cap Pair(n_1, n_2) \notin C_{g\_truth}$$

Here, $Pair(n_1, n_2)$ denotes a pair of nodes (i.e., lines of code snippets), C_{CoRA} denotes the cluster division by *CoRA*, and C_{g_truth} denotes the cluster division by the ground truth. And we calculate the Accuracy as follows:

$$Accuracy = \frac{2(\#TP + \#TN)}{n(n-1)} \tag{4}$$

Furthermore, we explore the influence of different edges in the clustering. We will explain the details later.

For RQ2, to evaluate how well importance ranking result matches the ground truth, we use the Mean Average Precision [30] and the Mean Reciprocal Rank [31] as our evaluation metric, which are both widely-used evaluation metrics in information retrieval. In particular, we calculate the MAP as follows:

$$MAP = \frac{\sum_{k=1}^{C} \frac{\sum_{i=1}^{N} rank(i)}{N}}{C} \tag{5}$$

Here, C denotes the number of clusters in a commit, N denotes the number of nodes (i.e., lines of code snippets) in a cluster, $rank(i)$ denotes the importance ranking by *CoRA*. For example, consider that a cluster has 3 nodes, and they are ranked at 1,2,3 by the ground truth, and they are ranked at 1,3,2 by *CoRA* respectively. The average precision score of this cluster is: $\frac{1/1+2/3+2/3}{3} = 77.8\%$. Then we calculate the MRR as follows:

$$MRR = \frac{\sum_{k=1}^{C} \frac{1}{first(i)}}{C} \tag{6}$$

Here, $first(i)$ denotes the importance ranking of the most important node (i.e., node's importance rank at 1st position in ground truth) in a cluster by *CoRA*. C denotes the number of clusters in a commit. For example, consider that a commit has three clusters, and their most important nodes are ranked at 1,3,5 respectively. The MRR of this commit is: $\frac{1/1+1/3+1/5}{3} = 51.1\%$.

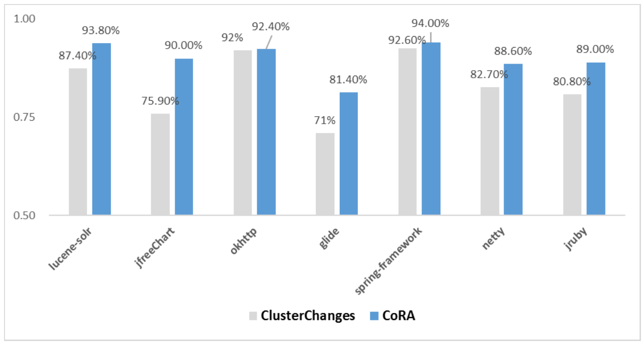

Fig. 3. Accuracy of *CoRA* and ClusterChanges for 7 projects, 50 commits.

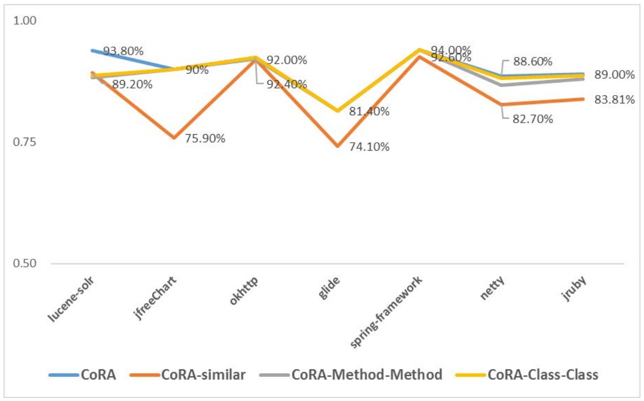

Fig. 4. Accuracy of *CoRA* without different edges for 7 projects, 50 commits.

3) **Qualitative Experiments Design:** We mainly conduct qualitative experiments for RQ3.

For RQ3, we are concerned about the quality of the description generated by *CoRA*, based on the three properties: *content adequacy, conciseness,* and *expressiveness. Content adequacy* judges whether the description contains almost important information about the commit. *Conciseness* assesses whether the description is concise, i.e., whether the description contains superfluous and useless information. *Expressiveness* evaluates whether the description is easy to understand. We evaluated the quality of a property in description by adopting a 3-points Likert Scale similarly to [6] [27]. In particular, we invited eight participants who established the ground truth to judge the descriptions, because they are familiar with these commits. Then we asked participants some questions (see Table V) by using an online tool.

B. Results & Analysis

Table VI presents the statistics of the tangled commit decomposition by *CoRA*. We obtained the code change graph after commit decomposition. On average, the number of nodes (i.e., fine-grained code changes) we extracted is 105, the number of code dependencies (i.e., *Def-Usage edges, method-method edges and class-class edges*) we established is 125. And we counted that the number of similar code changes in a commit is 106 on average. It indicates that similar code changes are very frequent in commits, so we should take advantage of the similarity between code changes when clustering code changes. Finally, we can cluster a tangled commit into about 5 independent clusters. However, in the vast majority of commits, there are always some fine-grained code changes that have no relationship or similarity with any other code changes, which are called trivial changes. We counted that the number of trivial changes in a commit is 14.7 on average.

1) **RQ1***:* To evaluate the accuracy of commit decomposition, we calculated the Rand Index value for total 50 commits from 7 projects. Fig. 3 shows the results. It can be seen that *CoRA* can achieve an average of 90% clustering accuracy, which is much higher than the average accuracy of 83.7% achieved by ClusterChanges [9], and *CoRA* can improve accuracy by 15% over the state-of-art practice on some projects. It means that *CoRA* can significantly improve the accuracy of fine-grained code changes clustering.

In order to explore the impact of different code dependencies and code similarity relationships on commit decomposition and clustering (i.e., we are concerned about why *CoRA* can significantly improve the accuracy), we conducted a series of comparative experiments by removing these edges (i.e., *method-method edges, class-class edges, similar edges* respectively). Fig. 4 shows the experimental results. It can be seen that removing any kind of edges from *CoRA* will reduce accuracy to varying degrees. Among them, the similarity between the code changes will significantly affect the clustering accuracy compared to other dependencies between the code changes. It means that considering the similarity between code changes in clustering is significant.

2) **RQ2***:* To best of our knowledge, we are the first to study the importance ranking of the fine-grained code changes. So we manually established the ground truth on the 50 commits (see part A of Section IV.), and we evaluate the accuracy of the ranking using MAP and MRR, which are both standard evaluation metrics in recommendation system and information retrieval. Table VII shows the results. It can be seen that our approach can achieve an average of 87.7% on the MAP metric. It means that our importance ranking algorithm works well on the task of fine-grained code changes ranking. Moreover, as for the maximum MAP value, the results show that our approach can achieve 100% maximum ranking accuracy in some commits in each open source project. We observed that this is because some commits are relatively simple, and our heuristic rules and the PageRank algorithm can identify their importance. As for the MRR value, we only consider the ranking accuracy of the most important fine-grained code changes (i.e., the 1st position code changes in ground truth). The results show that our approach can achieve an average of 85.3% value on the MRR metric. It means that our approach can identify the most important code changes in a cluster in

Table VI. Statistics of the Tangled Commits Decomposition by *CoRA*.

Projects	Commits	Nodes			Code- Dependency			Similar- Changes			Trivial-Changes			Clusters		
		Max	Min	Ave	Max	Min	Ave	Max	Min	Ave	Max	Min	Ave	Max	Min	Ave
lucene-solr	11	193	4	57	349	0	58	112	0	25	24	1	6.5	7	1	3
jfreeChart	5	589	21	191	626	4	156	744	0	400	103	4	25.4	13	2	7
okhttp	6	219	14	110	720	1	289	294	0	108	15	5	8.8	4	1	2.5
glide	5	251	37	152	439	1	132	404	4	97	61	10	36	13	3	7
spring-framework	7	220	12	120	636	1	148	1208	0	207	29	2	10	9	1	5
netty	6	225	25	110	417	11	139	82	2	29	25	7	15.5	5	2	4
jruby	10	392	13	77	243	4	55.5	148	0	29	80	0	14	37	1	6.6
Total	**50**	**589**	**4**	**105.4**	**720**	**0**	**124.7**	**1208**	**0**	**106.4**	**103**	**0**	**14.7**	**37**	**1**	**4.9**

Table VII. MAP and MRR Results of Fine-grained Code Changes Importance Ranking by *CoRA*.

Projects	Commits	MAP			MRR		
		Max	Min	Ave	Max	Min	Ave
lucene-solr	11	1	0.63	0.895	1	0.5	0.893
jfreeChart	5	1	0.705	0.889	1	0.667	0.850
okhttp	6	1	0.338	0.754	1	0.09	0.686
glide	5	1	0.844	0.923	1	0.806	0.897
spring-framework	7	1	0.417	0.877	1	0.25	0.826
netty	6	1	0.627	0.844	1	0.571	0.852
jruby	10	1	0.83	0.96	1	0.8	0.968
Total	**50**	**1**	**0.338**	**0.877**	**1**	**0.09**	**0.853**

the vast majority of cases. It is worth noting that the minimum MRR value is only 0.09%. This is due to the complexity of the fine-grained code changes in a cluster, but this is very rare.

3) **RQ3***:* We collected user feedback from eight participants who made human judgments on these questions in Table V using a 3-points Likert Scale. Each participant made judgments on 3 commits' descriptions. Fig. 5 presents the results.

As for content adequacy, participants agree with the answer 1 on 83.3% commits, i.e., most participants think that description by *CoRA* does not miss any important information of commits. However, there is still one participant who believes that our description misses some important information, i.e., the intent of the code changes, which is often mentioned in the commit message. For this, we believe that the intent of the code changes is undoubtedly important for understanding the code changes, but in our work, we are more concerned about the internal logical connection between these code changes and behavioural information of the commit, and we think this part of the information is more cumbersome and more valuable to summarize.

As for conciseness, most of the participants think that description has no unnecessary information. Participants agree with the answer 3 on 8.3% commits, they think that description has a lot of unnecessary information because it sometimes provides too much trivial code details, which developers may not care about. It indicates that our approach still need improvements. But in most cases, we can eliminate most of the unnecessary information by using our code importance ranking algorithm.

As for expressiveness, participants agree with the answer 1 on 70.8% commits, i.e., they think that our description is

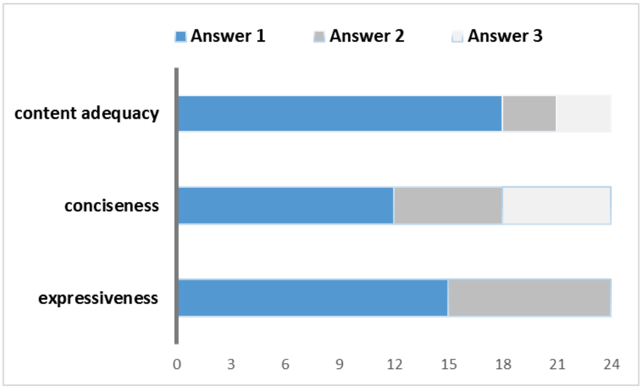

Fig. 5. Results to the questionnaire from user feedback.

easy to read and understand. And another participants suggest that "our tool should provide a better visualization tool to demonstrate the relationships between code changes", which should be considered in the future work.

C. Threats to validity

Internal validity Threats to internal validity may come from the process of establishing ground truth. Considering the complexity of tangled commits, manual labeling code clustering is a time-consuming and labor-intensive work and we understand that such a process is subject to mistakes. To address it, on the one hand, we have introduced third-party inspections to resolve different conflicting views. On the other hand, we preliminarily cluster the tangled commit to reduce the workload and complexity of judging fine-grained code changes by using CLDIFF [24].

External validity Threats to external validity mainly come from two aspects. First, we evaluated the clustering accuracy and importance ranking accuracy with a total number of 50 commits, which may be not very large scale. So could the results be generalized? To address it, we randomly selected commits from 7 open source projects, which have different scales and come from different domains, so we believe our datasets are generic and objective. Second, our human study results might have been affected by the differences between individual participants. To address it, we used the way of grouping participants to minimize the subjective impact of human judgment.

V. Related Work

We discuss related work from two aspects: the first is decomposing code changes; the second is describing code changes with natural language.

A. Code Changes Decomposition

Herzig and Zeller [8] first proposed that developers frequently group separate changes into a single commit, resulting in tangled code changes when committing changes. Ram et al. [21] also conducted an empirical investigation on code change reviewability and they proved that composite changes will make code review difficult. And Tao et al. [1] also presented an exploratory study to indicate that developers need to decompose changes in order to understand them better, which is one of the principal motivations of our work.

There have been some prior practices towards the challenge of decomposing tangled commits. Kirinuki et al. [32] utilized the Longest Common Subsequence algorithm to identify the tangled code changes by comparing previous code changes to the one being committed. Herzig and Zeller [8] proposed a multi-predictor approach to untangle code changes by using information such as file distance and call graph of the code changes. Similarly, Tao and Kim [12] proposed a heuristic-based approach to automatically partition composite changes, such that each sub-change in the partition is more cohesive and self-contained. Barnett et al. [9] introduced the ClusterChanges, an automatic technique for decomposing changesets, which utilize the *def-use* information. Dias et al. [11] proposed the EpiceaUntangler, which leverage machine learning and clustering to decompose tangled code changes by relying on the code changes information (e.g., code structure, message sending and variable accessing) during development. Zhou et al. [33] proposed the INFOX for identifying features in forks, their approach clusters cohesive code fragments using code and network-analysis (i.e., community detection algorithm) techniques.

An alternative approach to decompose large changes is to cluster and order their parts. Baum et al. [53] have proposed a theory on how to conduct this ordering and Spadini et al. [52] have conducted an experiment showing that the order in which test code is presented with respect to production code changes the effectiveness of the code review.

B. Code Changes Description

Automatic summarization of software artifacts with natural language has been widely investigated by researchers [27] [41] [42] and most of the existing techniques work mainly on code changes description [3] [4] [5] [6] [7] [10] [28] [40]. Originally, researchers proposed the line-based program analysis approach to summarize the differences in a commit. Jackson and Ladd [34] designed a *Semantic Diff* tool that takes two versions of a procedure and generates a report describing the semantic differences between them. Similarly, other practices that improve line-based differencing tools are *LDiff* by Canfora et al. [35] and *iDiff* by Nguyen et al. [36]. Recently, researchers proposed the tree-based program analysis approach to describe the code differences [23] [24] [43] [44]. Fluri et al. [43] proposed ChangeDistiller, a tree differencing algorithm for fine-grained source code changes extraction. Based on the ideas of ChangeDistiller, Dotzler and Philippsen [44] proposed a novel move optimized tree differencing algorithm that has a higher accuracy in detecting moved code parts. Huang et al. [24] proposed the ClDiff to generate concise linked code differences whose granularity is in between the existing code differencing and code change summarization methods based on existing work GumTree [23].

Other strategies have been explored to describe code changes, including program static analysis and machine learning. Buse and Weimer [5] introduced the DeltaDoc, which captures the behavioral changes for every method and the conditions under which they occur by using symbolic execution and summarization techniques. ChangeScribe [6] [7] is designed to generate commit messages by taking into account commit stereotype, changes type and the impact set of the underlying changes. Jiang et al. [10] adapted Neural Machine Translation to automatically "translate" diffs into commit messages. Liu et al. [4] performed an in-depth analysis of the NMT approach by Jiang et al. [10] and they proposed a simpler and faster approach, named *NNGen* to generate concise commit messages using the nearest neighbor algorithm.

Some work utilized related software artifacts (e.g., bug reports or mail-listings) to describe code changes. Rastkar and Murphy [28] proposed the use of multi-document summarization and machine learning-based techniques to generate a concise natural language description of why code changed. Similarly, Moreno et al. [37] introduced the ARENA, an approach for the automatic generation of release notes. ARENA extracts changes from the source code, summarizes them, and integrates them with information from versioning systems and issue trackers.

VI. Conclusion

This paper introduced *CoRA*, a novel approach for automatically decomposing and describing tangled code changes. *CoRA* first extracts fine-grained code changes from a tangled commit, then it clusters these code changes using code change graph and mining similar code changes. Next, *CoRA* identifies the importance ranking of these code changes for eliminating trivial changes, and then describes each cluster based on pre-defined natural language templates. *CoRA* has been evaluated with a total number of 50 commits from seven different projects, aimed at accessing the clustering accuracy, importance ranking accuracy of code changes, quality of the description and the usefulness of *CoRA*.

Acknowledgment

We are grateful to Alberto Bacchelli for his insightful comments and constructive suggestions.

This paper is supported by the National Natural Science Fund for Distinguished Young Scholars (Grant No. 61525201) and the National Key Research and Development Project of China (Grant No. 2016YFB1000800).

References

[1] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim.2012. How Do Software Engineers Understand Code Changes?:An Exploratory Study in Industry. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.51:151:11

[2] Pawlik M, Augsten N. Tree edit distance: Robust and memory-efficient[J]. Information Systems, 2016, 56: 157-173.

[3] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 422431.

[4] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine translation-based commit message generation: how far are we? In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 373384, 2018.

[5] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 3342.

[6] Luis Fernando Corts-Coy, Mario Linares-Vsquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On Automatically Generating Commit Messages via Summarization of Source Code Changes. In IEEE International Working Conference on Source Code Analysis and Manipulation. 275284.

[7] Mario Linares-Vsquez, Luis Fernando Corts-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, Vol. 2. IEEE, 709712.

[8] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In Proceedings of the 10th Working Conference on Mining Software Repositories. 121130.

[9] Mike Barnett, Christian Bird, Joo Brunet, and Shuvendu K.Lahiri. 2015. Helping Developers Help Themselves:Automatic Decomposition of Code Review Changesets. In Proceedings of the 37th International Conference on Software Engineering Volume 1.134144.

[10] Siyuan Jiang and Collin McMillan. 2017. Towards automatic generation of short summaries of commits. In Proceedings of the 25th International Conference on Program Comprehension. IEEE Press, 320323.

[11] Dias, M., Bacchelli, A., Gousios, G., Cassou, D., and Ducasse, S. (2015, March). Untangling fine-grained code changes. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER) (pp. 341-350). IEEE.

[12] Tao, Y., and Kim, S. (2015, May). Partitioning composite code changes to facilitate code review. In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (pp. 180-190). IEEE.

[13] Kawrykow, D., & Robillard, M. P. (2011, May). Non-essential changes in version histories. In 2011 33rd International Conference on Software Engineering (ICSE) (pp. 351-360). IEEE.

[14] Rahman, Mohammad Masudur, and Chanchal K. Roy. Improved query reformulation for concept location using CodeRank and document structures[C]//Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, 2017: 428-439.

[15] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. in Proceedings of the Third International Workshop on Predictor Models in Software Engineering, ser. PROMISE 07. IEEE Computer Society, 2007.

[16] A. Frank Ackerman, Priscilla J. Fowler, and Robert G. Ebenau. Software inspections and the industrial production of software. in Proc. of a symposium on Software validation: inspection-testing-verification-alternatives, 1984, pp. 13–40.

[17] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. Software inspections: an Effective Verification Process. IEEE Software, 1989.

[18] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. in Proceedings of the 35th International Conference on Software Engineering, 2013.

[19] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German. Open Source Peer Review Lessons and Recommendations for. IEEE Software, 2012.

[20] Peter C Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.ACM,202212.

[21] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What makes a code change easier to review: an empirical investigation on code change reviewability. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2018.

[22] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. Comput. Netw. ISDN Syst., 30(1-7):107117, 1998.

[23] Jean-Rmy Falleri, Floral Morandat, Xavier Blanc, Matias Martinez, Martin Monperrus. Fine grained and Accurate Source Code Differencing. Proceedings of the International Conference on Automated Software Engineering,2014,Vsteras,Sweden. pp.313-324,2014.

[24] Huang, K., Chen, B., Peng, X., Zhou, D., Wang, Y., Liu, Y., and Zhao, W. (2018, September). ClDiff: generating concise linked code differences. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (pp. 679-690). ACM.

[25] A. Alon and M. Tennenholtz. Ranking Systems: The PageRank Axioms. In Proceedings of the 6th ACM conference on Electronic commerce, 2005, pp. 1-8.

[26] A. N. Langville and C. D. Meyer, Googles PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press. ISBN 0-691-12202-4. 2006.

[27] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. VijayShanker. Automatic generation of natural language summaries for java classes. In ICPC13, pages 2332, 2013.

[28] Rastkar, S., & Murphy, G. C. (2013, May). Why did this code change?. In Proceedings of the 2013 International Conference on Software Engineering (pp. 1193-1196). IEEE Press.

[29] Fahad, A. , Alshatri, N. , Tari, Z. , Alamri, A. , Khalil, I. , and Zomaya, A. Y. , et al. (2014). A survey of clustering algorithms for big data: taxonomy and empirical analysis. IEEE Transactions on Emerging Topics in Computing, 2(3), 267-279.

[30] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schtze. Introduction to information retrieval. Vol. 1, no. 1. Cambridge: Cambridge university press, 2008.

[31] Voorhees, Ellen M. The TREC-8 Question Answering Track Report. In Trec, vol. 99, pp. 77-82. 1999.

[32] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. Hey! are you committing tangled changes? in Proceedings of the 22Nd International Conference on Program Comprehension, 2014, pp. 262265.

[33] Zhou, S., Stanciulescu, S., Leenich, O., Xiong, Y., Wasowski, A., and Kstner, C. (2018, May). Identifying features in forks. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE) (pp. 105-116). IEEE.

[34] D. Jackson and D. Ladd. Semantic diff: A tool for summarizing the effects of modications. In ICSM94, pages 243252, 1994.

[35] G. Canfora, L. Cerulo, and M. D. Penta. Ldiff: An enhanced line differencing tool. In ICSE09, pages 595 598, 2009.

[36] H. A. Nguyen, T. T. Nguyen, H. V. Nguyen, and T. N. Nguyen. idiff: Interaction-based program differencing tool. In ASE11, pages 575575, 2011.

[37] Laura Moreno,Gabriele Bavota,Massimiliano Di Penta,Rocco Oliveto,Andrian Marcus,and Gerardo Canfora. 2017. ARENA:an approach for the automated generation of release notes. IEEE Transactions on Software Engineering 43, 2 (2017),106127.

[38] T. Zimmermann, P. Weigerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. in Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, May 2004, pp. 563572.

[39] P. L. Li, R. Kivett, Z. Zhan, S.-e. Jeon, N. Nagappan, B. Murphy, and A. J. Ko. Characterizing the differences between pre- and post-release versions of software. in Proceeding of the 33rd international conference on Software engineering. ACM, 2011, pp. 716725

[40] M. Asaduzzaman, C. K. Roy, K. A. Schneider and M. D. Penta. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. 2013 IEEE International Conference on Software Maintenance, Eindhoven, 2013, pp. 230-239.

[41] McBurney, P. W., & McMillan, C. (2014, June). Automatic documentation generation via source code summarization of method context. In Proceedings of the 22nd International Conference on Program Comprehension (pp. 279-290). ACM.

[42] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., & Vijay-Shanker, K. (2010, September). Towards automatically generating summary com-

ments for java methods. In Proceedings of the IEEE/ACM international conference on Automated software engineering (pp. 43-52). ACM.

[43] Fluri, B., Wuersch, M., PInzger, M., & Gall, H. (2007). Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on software engineering, 33(11), 725-743.

[44] Dotzler, G., & Philippsen, M. (2016, September). Move-optimized source code tree differencing. In 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 660-671). IEEE.

[45] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. in Proceedings of the 30th international conference on Software engineering. ACM, 2008, pp. 541550.

[46] D. M. German, G. Robles, G. Poo-Caamano, X. Yang, H. Iida, and K. Inoue. Was my contribution fairly reviewed?: a framework to study the perception of fairness in modern code reviews. in Proceedings of the 40th International Conference on Software Engineering. ACM, 2018, pp. 523534

[47] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. Software Engineering,IEEE Transactions on,vol.35, no. 5, pp. 684702, 2009

[48] H. C. Benestad, B. Anda, and E. Arisholm. Understanding software maintenance and evolution by analyzing individual changes: a literature review. Journal of Software Maintenance and Evolution: Research and Practice, vol. 21, no. 6, pp. 349378, 2009.

[49] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality:how developers see it. In Proceedings of the 38th International Conference on Software Engineering. ACM,10281038.

[50] M. Chilowicz, E. Duris and G. Roussel. Syntax tree fingerprinting for source code similarity detection. 2009 IEEE 17th International Conference on Program Comprehension, Vancouver, BC, 2009, pp. 243-247.

[51] I. D. Baxter, A. Yahin, L. Moura,M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In ICSM '98, page 368, Washington, DC, USA, 1998. IEEE-CS.

[52] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, Alberto Bacchelli. Test-driven code review: an empirical study. Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019.

[53] Baum T, Schneider K, Bacchelli A. On the optimal order of reading source code changes for review[C]//2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017: 329-340.