CLDIFF: Generating Concise Linked Code Differences

Kaifeng Huang*
School of Computer Science
Fudan University, China

Bihuan Chen†
School of Computer Science
Fudan University, China

Xin Peng
School of Computer Science
Fudan University, China

Daihong Zhou
School of Computer Science
Fudan University, China

Ying Wang
School of Computer Science
Fudan University, China

Yang Liu
Nanyang Technological University,
Singapore

Wenyun Zhao
School of Computer Science
Fudan University, China

ABSTRACT

Analyzing and understanding source code changes is important in a variety of software maintenance tasks. To this end, many *code differencing* and *code change summarization* methods have been proposed. For some tasks (e.g. code review and software merging), however, those differencing methods generate too fine-grained a representation of code changes, and those summarization methods generate too coarse-grained a representation of code changes. Moreover, they do not consider the relationships among code changes. Therefore, the generated differences or summaries make it not easy to analyze and understand code changes in some software maintenance tasks.

In this paper, we propose a code differencing approach, named CLDIFF, to generate concise linked code differences whose granularity is in between the existing code differencing and code change summarization methods. The goal of CLDIFF is to generate more easily understandable code differences. CLDIFF takes source code files before and after changes as inputs, and consists of three steps. First, it pre-processes the source code files by pruning unchanged declarations from the parsed abstract syntax trees. Second, it generates concise code differences by grouping fine-grained code differences at or above the statement level and describing high-level changes in each group. Third, it links the related concise code differences according to five pre-defined links. Experiments with 12 Java projects (74,387 commits) and a human study with 10 participants have indicated the accuracy, conciseness, performance and usefulness of CLDIFF.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

Code Differencing, Program Comprehension, AST

ACM Reference Format:
Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. 2018. CLDIFF: Generating Concise Linked Code Differences. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238219

1 INTRODUCTION

Analyzing and understanding source code changes is important in a variety of software maintenance tasks. For example, to improve software quality, developers often spend a significant amount of time to comprehend code changes during code review [6, 52]; to resolve merging conflicts, code change knowledge is required during software merging [43]; and to efficiently find regression bugs, code change information is useful for selecting the test cases that need to be rerun during regression testing [51]. Therefore, a number of *code differencing* and *code change summarization* methods have been proposed to represent code changes at different granularity.

In particular, for code differencing, text-based methods [4, 9, 44, 46, 50] are unaware of the syntactic structure of source code and compute textual differences that are not easy for further analysis and understanding. Instead, tree-based methods [16, 17, 19, 21, 24] directly work at the abstract syntax tree (AST) granularity for generating fine-grained syntactic code differences. The differences between two ASTs are in the form of an edit script, a sequence of *edit actions* to transform the AST before changes to the AST after changes. Such edit scripts can be too fine-grained, too scattered, and too long to understand code changes in some applications (e.g. code review and software merging), especially for large code changes [24]. Moreover, the relationships among code changes (e.g. a change to the signature of a method can result in changes to all the invocations of the method) are missing, which are in fact important for code change analysis and understanding (e.g. the related code changes need to be considered together during code review or software merging).

On the other hand, code change summarization methods [27, 37, 38, 45, 49] generate natural language summaries to describe code changes, e.g. the motivation behind code changes [49], the commit

*K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang and W. Zhao are also with the Shanghai Key Laboratory of Data Science, Fudan University, China and the Shanghai Institute of Intelligent Electronics & Systems, China.
†B. Chen is the corresponding author.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238219

Figure 1: An Example of Code Changes from Commit `3c1adf7` in spring-framework

message for code changes in a commit [27, 37, 38], and the release note for code changes in a release [45]. These methods are mostly developed for the ease of documentation of code changes. Thus, the generated summaries are usually too coarse-grained to be useful for in-depth analysis and understanding of code changes (e.g. code review and software merging).

To address the problems with existing methods and to provide more easily understandable code differencing information required for tasks such as code review and software merging, we propose and implement a novel code differencing approach, named CLDIFF. It is designed to generate a concise, linked representation of code differences, whose granularity is in between the existing code differencing and code change summarization methods. In other words, CLDIFF not only generates short and informative code differences, but also establishes their relationships.

Technically, CLDIFF takes as inputs source code files before and after changes (e.g. in a patch, commit or release), and works in three steps. First, CLDIFF pre-processes the source code files by pruning unchanged declarations from parsed ASTs. The purpose is to avoid unnecessary differencing analysis on unchanged AST elements in the second step. Second, CLDIFF generates concise code differences via grouping the fine-grained code differences, generated by GUMTREE [17], at or above the statement level and describing high-level changes in each group. The underlying idea is to put together the fine-grained code differences that are scattered but related to a high-level AST element. Third, CLDIFF links the related concise code differences according to five pre-defined links. The motivation is to consider such related code changes as a whole in some tasks.

We have implemented CLDIFF for Java, and conducted experiments with 12 open-source Java projects (i.e. 74,387 commits in total) to evaluate the accuracy, conciseness and performance of CLDIFF as well as a human study with 10 participants to evaluate the usefulness of CLDIFF. The results have demonstrated that CLDIFF generated concise code differences and established their links with an accuracy of 99% and 98%, respectively; and compared to GUMTREE, CLDIFF generated more than 80% shorter edit script for 48% commits with 72% shorter time, and was more useful in change understanding.

In summary, this work makes the following contributions.

- We proposed a code differencing approach named CLDIFF to generate concise linked code differences.
- We implemented CLDIFF for Java, and provided visualization for the generated concise linked code differences.
- We conducted experiments with 12 open-source Java projects as well as a human study with 10 participants to demonstrate CLDIFF's accuracy, conciseness, performance and usefulness.

2 PRELIMINARIES

AST. A source code file can be parsed into an abstract syntax tree (AST), which is a rooted, labeled, ordered tree. Each node has a *label* to indicate its type representing a structural element (e.g. declaration) of the source code. Some nodes have a string *value* to indicate the actual token (e.g. variable name) in code.

Example 2.1. Fig. 2(a) and 2(b) give the two ASTs before and after the code changes at Line 7–12 in Fig. 1. We only show partial ASTs for clarity. The AST in Fig. 2(a) contains eight nodes. Specifically, node n_5 has three child nodes n_6, n_7 and n_8, and its label is MethodInvocation. The label of n_6, n_7 and n_8 is SimpleName. n_6, n_7 and n_8 respectively denote the receiver, name and argument of the method invocation; and their value is taskDecorator, decorate and command.

AST Node Type Hierarchy. The type of the root node of an AST is CompilationUnit, whose child nodes can be of the type BodyDeclaration. The common subtypes of BodyDeclaration are TypeDeclaration (class or interface declaration), MethodDeclaration (method or constructor declaration), Initializer (static or instance initializing block), FieldDeclaration (field declaration), and EnumDeclaration (enumeration declaration). Declarations can contain a list of statements which have 22 different statement types (e.g. IfStatement and VariableDeclarationStatement). Statements can contain a list of expressions (e.g. MethodInvocation). Therefore, declaration, statement and expression have a decreasing granularity. However, they can be nested with each other.

AST Differencing. Given two ASTs before and after code changes (i.e. AST_b and AST_a), AST differencing tools can generate an *edit*

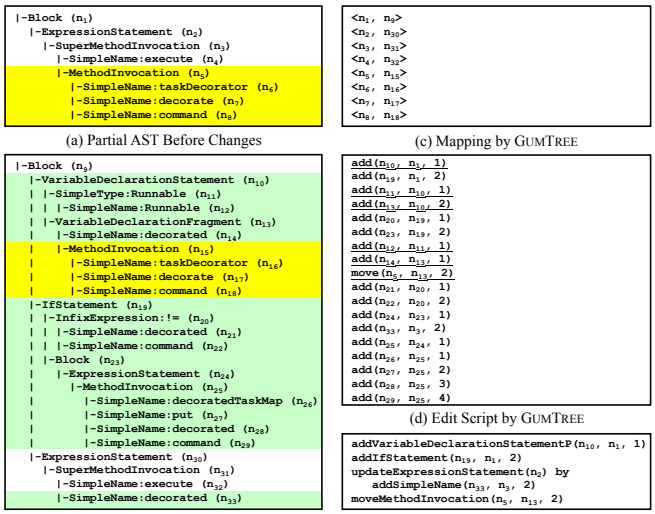


Figure 2: An Example of Differencing on Line 7–12 in Fig. 1

script (i.e. a sequence of *edit actions*). By sequentially applying the edit actions, we can convert AST_b to AST_a. Here we apply the state-of-the-art tool, GUMTREE [17], to generate fine-grained code differences. GUMTREE works in two steps. First, it uses heuristics to derive a *mapping* between nodes in two ASTs. The mapping is a set of pairs $\langle n_b, n_a \rangle$, where node n_b in AST_b is mapped to node n_a in AST_a. Then, based on the mapping, it generates the edit script that contains four kinds of edit actions, i.e. *update*, *add*, *delete* and *move*.

- $update(n, v)$ replaces the value of node n with a value v.
- $add(n, p, i)$ adds a new node n as the i-th child node of node p if p is not null. Otherwise, n becomes the new root node and has the previous root node as its only child node.
- $delete(n)$ removes a leaf node n.
- $move(n, p, i)$ moves node n to be the i-th child node of node p. All descendant nodes of n are moved together with n.

Example 2.2. Fig. 2(c) give the mapping, generated by GUMTREE, between the nodes in the two ASTs in Fig. 2(a) and 2(b). Here all the eight nodes in Fig. 2(a) are mapped. Based on this mapping, GUMTREE generates an edit script containing 18 edit actions, as listed in Fig. 2(d). Specifically, one of the edit actions is $move(n_5, n_{13}, 2)$, which moves the method invocation rooted at n_5 to be the second child node of a variable declaration fragment rooted at n_{13}.

3 MOTIVATION AND OVERVIEW

In this section, we motivate the proposed approach with an example before introducing our approach overview.

3.1 Motivation Example

Fig. 1 lists three source code files changed in a commit taken from spring-framework. In class ①, a *for* structure (Line 2–4) is added, where a newly-declared method (Line 5) is invoked. This new method is then overridden in both class ② (Line 13–14) and class ③ (Line 29–30) because ② and ③ inherit ①. In class ②, a field is declared (Line 6), a variable is extracted (Line 7–8), and both of them are used in a newly-added *if* structure (Line 9–11). In class ③, a filed is declared (Line 15) and then used in a newly-declared method (Line 28). This new method is then invoked in two similar code changes (Line 16–21 and 22–27). This example is used throughout the paper.

Given the code changes at Line 7–12 in class ② in Fig. 1, we present the two partial ASTs before and after the changes in Fig. 2(a) and 2(b). The added nodes are highlighted in green and the moved nodes are highlighted in yellow. Here no deletion or update is involved. For these changes, GUMTREE generates the edit script shown in Fig. 2(d), which means that 17 new nodes are added and one node is moved.

However, some edit actions (e.g. those underlined ones in Fig. 2(d)) are related to a high-level AST element (e.g. variable declaration statement), but are scattered across the edit script. Such related but scattered edit actions, although being exhibited together in visualization, make the follow-up analysis and understanding of code changes difficult. For example, in code review, developers will recognize the insertion of a variable declaration statement intuitively rather than thinking of the fine-grained tree operations. Similarly, in software merging, a newly-added variable declaration statement will be considered as a whole to resolve a conflict. Therefore, to generate more easily-understandable code differences for both developers and automatic analysis tools, we try to obtain high-level concise code differences at or above the statement level. Fig. 2(e) shows the edit script generated by our approach. It has four high-level edit actions, i.e. adding a variable declaration statement, adding an *if* statement, updating an expression statement by adding a simple name, and moving a method invocation to be a part of the newly-added variable declaration statement (see approach details in Section 4.2).

On the other hand, the relationships among code changes are not considered in GUMTREE but are actually helpful in the analysis and understanding of code changes. As an example, for the newly-declared method at Line 5 in Fig. 1, it is invoked at Line 3 and overridden at Line 13–14 and 29–30. As another example, the code changes at Line 16–21 are almost the same to the code changes at Line 22–27. Such relationships capture the causality of code changes, which can speed up the process of code review and improve the accuracy of merging conflict resolution. Therefore, we attempt to establish the links among generated high-level code differences (see approach details in Section 4.3).

3.2 Approach Overview

Fig. 3 presents an overview of CLDIFF. The inputs of CLDIFF are a set of pairs of source code files before and after changes (e.g. in a commit, patch or release). The outputs can be visualized by our web-based tool. CLDIFF works in three steps, pre-processing (Section 4.1), generating concise code differences (Section 4.2) and linking code differences (Section 4.3), to generate concise linked code differences.

First, since code changes often affect a small part of a source code file and a large amount of code remains unchanged, we pre-process the pairs of source code files to remove some unchanged code in order to avoid unnecessary differencing analysis. To this end, CLDIFF first parses every pair of source code files into an AST pair, and then prunes unchanged declaration-level elements from the AST pair based on a hashing technique. Here we select declaration as the pruning unit to strike a balance between feasibility and scalability.

Second, as fine-grained code differences (in the form of edit actions) are often related to high-level AST elements but scattered across the edit script, we generate high-level concise code differences at or

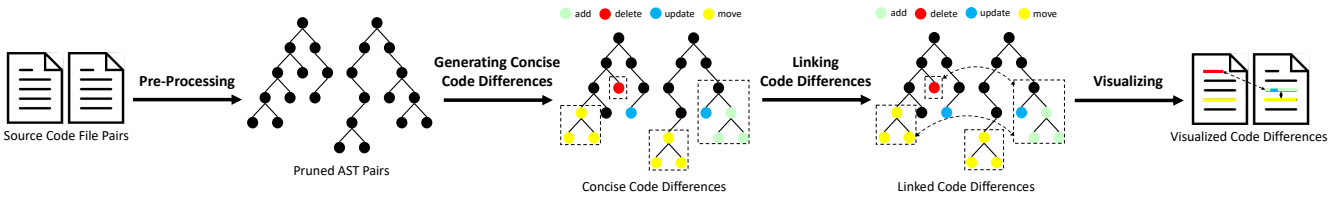


Figure 3: An Overview of CLDIFF

above the statement level. Specifically, CLDIFF first uses GUMTREE [17] to obtain the mapping and edit actions for each pruned AST pair. Then, it traverses the edit actions and the pruned AST pair to iteratively group edit actions that are related to an AST element at or above the statement level. Finally, it generates a concise code difference for each group to capture its high-level changes. Here we choose statement as the suitable granularity of code differences to better reflect developers' intuition about code changes.

Third, since code changes are often causally related with each other, we establish links among the generated concise code differences. Specifically, based on the concise code differences for each pair of source code files, CLDIFF checks whether there exists a code change link between two concise code differences according to five pre-defined links (e.g., *Def-Use* link).

4 METHODOLOGY

In this section, we elaborate each step of CLDIFF (Fig. 3) in detail. Our approach is general, although we explain our approach for Java.

4.1 Pre-Processing

In the first step, we pre-process the source code files to prune some unchanged declarations from parsed ASTs.

Given each pair of source code files $\langle f_b, f_a \rangle$, we parse it into an AST pair $\langle AST_b, AST_a \rangle$, where AST_b is the AST of the file f_b before code changes and AST_a is the AST of the file f_a after code changes. Then, we traverse AST_b to compute two hash values for the node whose label is a field, enumeration, method, inner class, or initializor declaration, and store the AST node to a map whose key is the two hash values. One hash value is calculated over the canonical name of the residing class and is used to distinguish the same declaration in both outer and inner classes. Another hash value is calculated over the corresponding declaration code (i.e. the subtree rooted at the node). Finally, we traverse AST_a to compute the two hash values for each declaration node, and prune the node (including all its descendant nodes) from both AST_b and AST_a if the two hash values find a match in the map. The output is a pruned AST pair $\langle AST'_b, AST'_a \rangle$. Notice that as comments and Javadocs are not treated as code, they are removed from ASTs beforehand.

4.2 Generating Concise Code Differences

In the second step, we generate concise code differences from fine-grained code differences. Our underlying idea is to put fine-grained code differences within a statement or declaration AST element to a group and describe high-level changes in the group.

Specifically, given a pruned AST pair $\langle AST'_b, AST'_a \rangle$, we use GumTree [17] to generate the mapping $\mathcal{M}$ and the edit script $\mathcal{A}$ between the two ASTs. Recall that $\mathcal{M}$ maintains the mapped AST node pairs and $\mathcal{A}$ stores the edit actions (Section 2). Then, we traverse the edit actions in three phases to group edit actions and generate concise code differences.

Phase 1. Different from *update*, *add* and *delete* actions that only affect one atomic node but not its descendant nodes, *move* actions move the whole subtree rooted at one node. Therefore, a *move* action can already reflect high-level concise code changes. In that sense, for each $move(n, p, i) \in \mathcal{A}$, we generate a concise code difference $moveX(n, p, i)$, where X is the label of node n and explicitly reflects the syntactic information, and remove $move(n, p, i)$ from $\mathcal{A}$.

Example 4.1. The edit script in Fig. 2(d) contains one *move* action $move(n_5, n_{13}, 2)$ that moves a whole method invocation. Thus, CLDIFF generates $moveMethodInvocation(n_5, n_{13}, 2)$.

Phase 2. Some statements or declarations have simple structures, while others contain complex ones with statements or declarations nested as composing elements. In that sense, an *add* or a *delete* action on a statement or declaration AST node is mostly accompanied by simultaneous *add* or *delete* actions on its composing elements; i.e. a whole or a part of a statement or declaration is added or deleted together. Hence, we group edit actions with respect to the composing elements of a statement or declaration, and distinguish whether a whole or a part of a composing element is added or deleted together.

Before introducing how to group edit actions, we first categorize all statements and declarations into two categories and define their *base* and *composing* elements.

- *C1.* This category includes statements and declarations whose child nodes $\mathcal{N}$ can contain statements or declarations, e.g. IfStatement, TryStatement, MethodDeclaration and TypeDeclaration. We define each node $n \in \mathcal{N}$ that is a non-block statement or a declaration as a composing element, each child node of the node $n \in \mathcal{N}$ which is a block statement as a composing element, and all the other nodes in $\mathcal{N}$ and their parent node as a base element.
- *C2.* This category contains statements and declarations whose child nodes do not contain statements or declarations, e.g. ExpressionStatement, VariableDeclarationStatement, ReturnStatement and FieldDeclaration. They are defined as a base element and do not have composing elements.

Example 4.2. In Fig. 2, n_{10} is a variable declaration statement that belongs to *C2*; and thus n_{10} and all its descendant nodes are considered as the base element of n_{10}. n_{19} is an *if* statement which belongs to *C1*; and hence n_{19}, n_{20}, n_{21}, n_{22} and n_{23} are considered as the base element of n_{19} (representing the wrapper of the *if* statement intuitively), while n_{24} and all its descendant nodes are considered as a composing element of n_{19} (indicating the statement in the *if* statement body). Similarly, the base element of a method declaration denotes the method with an empty body, while its composing elements represent the statements in the method body.

Then we introduce how to group edit actions. Specifically, for each $add(n, p, i) \in \mathcal{A}$ where n is a statement or declaration, we put this action to $\mathcal{B}$ which maintains the *add* actions on the base element, locate n on AST_a (because *add* actions are applied on AST_a), and traverse n's descendant nodes in a depth-first way while distinguishing base and composing elements. For the base element, for each traversed node m, if m is newly-added by an *add* action a, we group a to $\mathcal{B}$ and continue the traversal on m's child nodes; otherwise (m is not newly-added, i.e. there exists a match in $\mathcal{M}$ for m), we mark $\mathcal{B}$ as a *partial addition*, stop our traversal on m's child nodes, but continue the traversal on other nodes in the base element. After completing the traversal, if $\mathcal{B}$ is marked as a partial addition, we generate a concise code difference $addXP(n, p, i)$, where X is the label of n, P denotes partial addition, and n is the subtree resulting from the actions in $\mathcal{B}$, and remove $\mathcal{B}$ from $\mathcal{A}$; otherwise (the whole base element is newly-added), we traverse the composing elements to determine whether they are *all* newly-added. If yes, we store all these *add* actions to C, generate a concise code difference $addX(n, p, i)$, where X is the label of n and n is the subtree resulting from the actions in $\mathcal{B}$ and C, and remove $\mathcal{B}$ and C from $\mathcal{A}$. If not, we generate $addXP(n, p, i)$ and remove $\mathcal{B}$ from $\mathcal{A}$. Intuitively, if one whole statement or declaration is added, we generate one code difference; otherwise, we generate code differences on its base and composing elements separately.

On the other hand, for each $delete(n) \in \mathcal{A}$ where n is a statement or declaration, we traverse n on AST_b (as *delete* actions are applied on AST_b) in the same way as for *add* actions, and generate either $deleteXP(n)$ or $deleteX(n)$.

Example 4.3. When traversing the edit script in Fig. 2(d), we first analyze $add(n_{10}, n_1, 1)$, which adds a variable declaration statement that belongs to *C2*. We group it with $add(n_{11}, n_{10}, 1)$, $add(n_{13}, n_{10}, 2)$, $add(n_{12}, n_{11}, 1)$ and $add(n_{14}, n_{13}, 1)$ in $\mathcal{B}$. As $\mathcal{B}$ is marked as a partial addition, we generate the first code difference in Fig. 2(e). Then we analyze $add(n_{19}, n_1, 2)$, which adds an *if* statement of *C1*. We group it with $add(n_{20}, n_{19}, 1)$, $add(n_{21}, n_{20}, 1)$, $add(n_{22}, n_{20}, 2)$ and $add(n_{23}, n_{19}, 2)$ in $\mathcal{B}$. As $\mathcal{B}$ is not marked, we further group $add(n_{24}, n_{23}, 1)$ with $add(n_{25}, n_{24}, 1)$, $add(n_{26}, n_{25}, 1)$, $add(n_{27}, n_{25}, 2)$, $add(n_{28}, n_{25}, 3)$ and $add(n_{29}, n_{25}, 4)$ in C, and then generate the second code difference in Fig. 2(e) that adds a complete *if* statement.

Example 4.4. Fig. 4 shows another case of generating concise code differences. When traversing the edit script in Fig. 4(e), we first encounter $add(n_{15}, n_1, 1)$, which adds an *if* statement that belongs to *C1*. We group it with all the other *add* actions in Fig. 4(e) in $\mathcal{B}$. As $\mathcal{B}$ is not marked, we further analyze its composing elements. However, the composing element is not newly-added but moved. Thus, we generate the first code difference in Fig. 4(f), which actually adds a wrapper of an *if* statement.

Phase 3. After Phase 1 and Phase 2, the remaining actions in $\mathcal{A}$ are only *add*, *delete* and *update* actions on non-statement and non-declaration AST nodes. Given that some actions are applied within the same statement or declaration, we group such actions together with respect to their common ancestor statement or declaration. In particular, for each traversed $add(n, p, i) \in \mathcal{A}$, we locate n's closest ancestor node m that is a statement or declaration in AST_a, replace m with its mapping m' in AST_b using $\mathcal{M}$ if m' exists, and put $add(n, p, i)$ to a list Q_m that maintains all the actions applied within

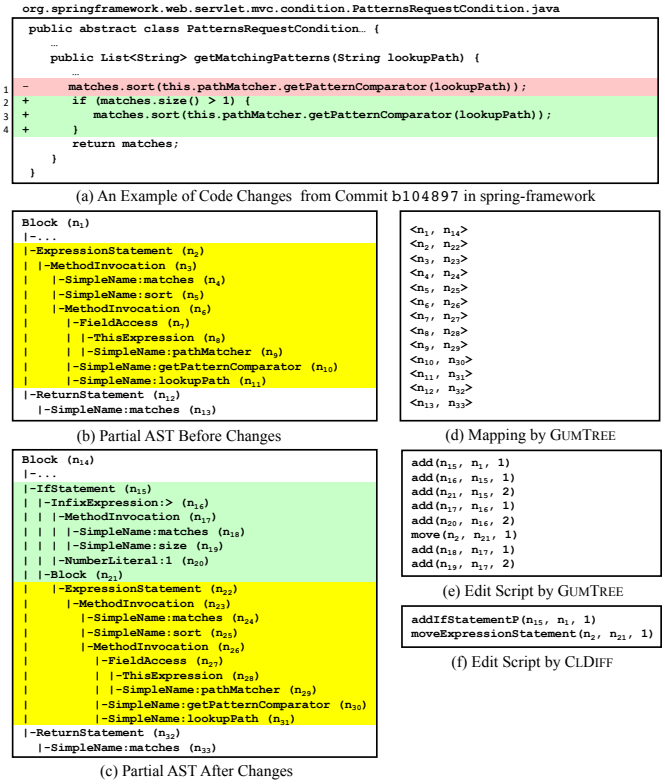


Figure 4: An Example of Concise Code Differences

m. Similarly, for each traversed $delete(n)$ or $update(n, v)$ in $\mathcal{A}$, we find n's closest ancestor node m that is a statement or declaration in AST_b, and store $delete(n)$ or $update(n, v)$ to Q_m. After the traversal, for each Q_m, we generate a concise code difference $updateX(m)$ *by* Y where X is the label of m and Y represents the actions in Q_m with the syntactic information highlighted in their action names. In this way, all originally-scattered edit actions on one statement or declaration are grouped together for the ease of analysis and understanding. Unlike our *add* and *delete* actions, m is not a subtree but an atomic node to inform that the actions in Q_m are applied on scattered descendant nodes of m.

Example 4.5. Following Example 4.1 and 4.3, there is only one remaining edit action $add(n_{33}, n_3, 2)$ in the edit script in Fig. 2(d) after Phase 1 and Phase 2. n_{33}'s closest ancestor node that is a statement or declaration in Fig. 2(b) is n_{30}, mapped to n_2 in Fig. 2(a). Hence, $updateExpressionStatement(n_2)$ *by* $addSimpleName(n_{33}, n_3, 2)$ is generated, as shown by the last code difference in Fig. 2(e).

4.3 Linking Code Differences

In the third step, we establish code change links among the generated concise code differences according to five pre-defined links. Such links reflect the causality of code changes.

We first define the five kinds of code change links, which are not meant to be exhaustive but to demonstrate that a small set of links are already useful in change understanding. They can be extended to incorporate new kinds of links.

- *Def-Use Link.* If the declaration of a variable, field or method is changed (i.e. added, deleted, updated or moved) by code difference

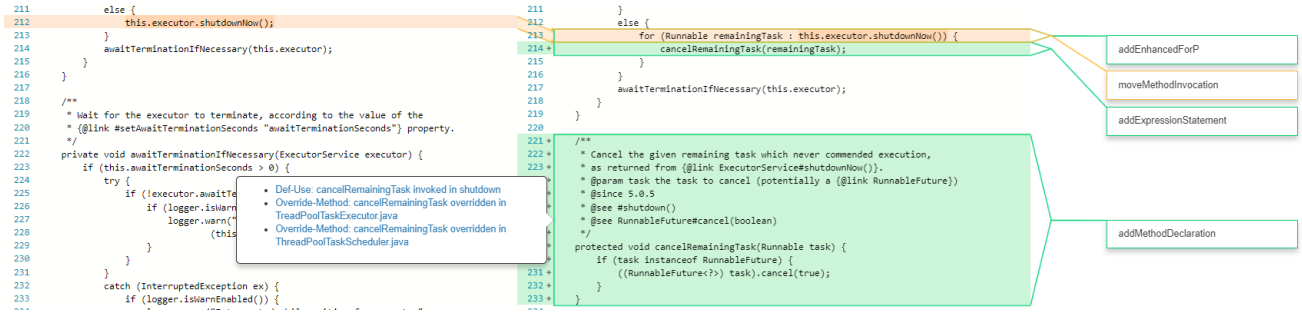


Figure 5: A Snapshot of Our Visualization Tool

d_1, the usage of the variable, field or method can be changed by d_2. We define the link between d_1 and d_2 as a *Def-Use* link $d_1 \xrightarrow{\text{DU}} d_2$.

- *Abstract-Method Link.* If the declaration of an abstract method in a class is changed by d_1, the implementation of the abstract method in each sub-class must be changed by d_2. We define the link between d_1 and d_2 as an *Abstract-Method* link $d_1 \xrightarrow{\text{AM}} d_2$.
- *Override-Method Link.* If the declaration of a method in a class is changed by d_1, the implementation of the method might be changed through override in each sub-class by d_2. We define the link between d_1 and d_2 as an *Override-Method* link $d_1 \xrightarrow{\text{OM}} d_2$.
- *Implement-Method Link.* If the declaration of a method in an interface is changed by d_1, the implementation of the method must be changed in each class that implements the interface by d_2. We define the link between d_1 and d_2 as an *Implement-Method* link $d_1 \xrightarrow{\text{IM}} d_2$.
- *Systematic-Change Link.* If two code differences d_1 and d_2 are similar, they might be caused by systematic changes (e.g. refactoring [35] and recurring bug fixes [47]). We define the link between d_1 and d_2 as a *Systematic-Change* link $d_1 \xleftrightarrow{\text{SC}} d_2$.

Then, we introduce how to establish these links based on concise code differences $\mathcal{D}_i$ for each pruned AST pair. Assuming that there are totally k AST pairs, i.e. $1 \leq i \leq k$. Specifically, to establish *Def-Use* links, we first find each $d \in \mathcal{D}_i$ that is applied on a variable declaration statement, a field declaration or a method declaration, and extract the name of the variable, field or method. Then, we locate every $e \in \mathcal{D}_i$ that is within the same scope (i.e. for a variable declaration statement, the scope is its enclosing method declaration; and for a field or method declaration, the scope is its enclosing class declaration) and involves a variable, field access or method invocation with the same name, and establish the link $d \xrightarrow{\text{DU}} e$. Here we only consider the *Def-Use* links within a limited scope; e.g. we do not consider that a method declaration might be used in another class.

To build *Abstract-Method*, *Override-Method* or *Implement-Method* links, we first find each $d \in \mathcal{D}_i$ that is applied on an abstract method declaration, a method declaration or an interface method declaration, and extract the method signature and the name of the enclosing abstract class, class or interface. Then, we find every $e \in \mathcal{D}_j$ $(j \neq i)$ that is applied on such a method declaration that it has the same method signature and its enclosing class extends a class or implements an interface with the same name, and construct the link $d \xrightarrow{\text{AM}} e$, $d \xrightarrow{\text{OM}} e$ or $d \xrightarrow{\text{IM}} e$.

To construct *Systematic-Change* links, for each *delete*, *add* or *move* action $d \in \mathcal{D}_i$ that is applied on node n_d, we first get each *delete*, *add* or *move* action $e \in \mathcal{D}_j$ $(e \neq d)$ that is applied on n_e whose label is the same to n_d. Then, we check whether the size of the grouped edit actions (see Section 4.2) for n_d and n_e is the same. If yes, we compute the bi-gram similarity [2] between the code snippets corresponding to the subtrees rooted at n_d and n_e. If the similarity is large than or equal to 0.8, we build the link $d \xrightarrow{\text{SC}} e$. For each *update* action, the overall procedure is similar but the similarity computation is different. Since our *update* actions often group a set of fine-grained edit actions that are scattered, n_d and n_e are atomic nodes. Hence, we get the subtrees rooted at n_d and n_e from the pruned AST pair (i.e. either from both AST_b and AST_a or only from AST_b depending on whether n_d and n_e can be respectively mapped in their $\mathcal{M}$), and compute the bi-gram similarity. Intuitively, this checks whether the changed code before and after changes is similar.

It is worth mentioning that our strategy of establishing links is designed to be heuristic and lightweight and directly work at the source code level, but not rely on heavyweight program analysis techniques. Our assumption is that code changes are often focused, and such a simple strategy is often sufficient to achieve a balance between accuracy and scalability. We leave it as our future work to investigate the cost-benefit of using heavyweight program analysis techniques to establish links.

Example 4.6. For the code changes in Fig. 1, CLDIFF correctly establishes all the links without any false positive or false negative. For example, it constructs a *Override-Method* links between the *addMethodDeclaration* for Line 5 and the *addMethodDeclaration* for Line 14. It establishes a *Def-Use* link between the *addVariableDeclarationStatementP* for Line 8 and the *addIfStatement* for Line 9–11. It builds a *Systematic-Change* link between the *updateVariableDeclaration* for Line 16, 19 and the *updateVariableDeclaration* for Line 22, 25.

5 IMPLEMENTATION AND EVALUATION

We have implemented CLDIFF for Java with 30K lines of Java code, and developed a web-based tool to visualize our concise linked code differences with 4.6K lines of JavaScript code. Fig. 5 gives a snapshot of our visualization tool. A concise code difference is visualized via highlighting the code and prompting the action name. A click on one of the highlighted code snippets will pop a window to show the links that are related to this code difference, while a click on one of the links will navigate to the corresponding code difference. CLDIFF is open-sourced and is available at [1].

Table 1: Projects Used in Our Experiments

Projects	Creation Date	LOC	Stars	Commits
RxJava	2012-03	270.0K	32.6K	4226
elasticsearch	2010-02	889.2K	30.5K	29929
okhttp	2011-05	60.0K	26.3K	2784
retrofit	2010-09	22.4K	27.5K	1090
spring-framework	2008-07	673.5K	20.7K	12838
zxing	2007-10	156.0K	18.3K	1793
netty	2008-08	258.6K	13.7K	11047
fastjson	2011-07	170.0K	13.3K	2304
guava	2009-06	342.0K	23.7K	3925
glide	2012-12	73.6K	21.4K	1745
mybatis-3	2010-05	96.0K	7.4K	1189
MPAndroidChart	2014-04	26.7K	21.8K	1517

5.1 Evaluation Setup

To evaluate the effectiveness of CLDIFF, we conducted experiments using 12 highly-stared open-source Java projects from GitHub by comparing CLDIFF with one of the state-of-the-art AST differencing tools, GUMTREE [17]. Table 1 reports the statistics about projects, including project name, creation date, lines of code, the number of stars, and the number of commits. The number of commits is computed by removing the commits that are not related to code changes (e.g. changes to configuration files) or only related to testing code changes. In total, 74,387 commits are used. We can see that these projects are all large-scale and popular, and have a long evolution history. This ensures that these projects contain rich and diverse code changes. GUMTREE was configured with the same setting as the one used in [17].

On the other hand, to evaluate the usefulness of CLDIFF, we conducted a human study with 10 participants to understand the changes in 10 commits. In particular, from our school, we hired 10 graduate students who had at least 2-years experience in Java programming. One of them had 6-years experience; and the average experience was 4 years. All the participants are not the authors of this paper. Besides, we randomly selected 10 commits from those 12 projects with the criterion that at most 6 Java source files were involved in a commit. This is to control the complexity of understanding code changes and thus keep the concentration of participants.

Using the previous setup, we conducted the experiments and the human study to answer the following research questions.

- **RQ1:** How is the accuracy of the generated concise code differences and the established links by CLDIFF? (Section 5.2)
- **RQ2:** How is the size of the generated concise code differences of CLDIFF compared to GUMTREE? (Section 5.3)
- **RQ3:** How is the performance overhead of CLDIFF compared to GUMTREE? (Section 5.4)
- **RQ4:** How is the usefulness of CLDIFF in understanding code changes compared to GUMTREE? (Section 5.5)

5.2 Accuracy Evaluation (RQ1)

To evaluate the accuracy of CLDIFF's generated concise code differences and established links, we randomly chose 10 commits from each project, and manually analyzed the results of CLDIFF on them. Table 2 shows the accuracy results, where we also reported the total number of generated code differences for the 10 commits and the total number of established links under column *Size*. In total, we analyzed 1,456 code differences, and achieved an accuracy of 99%; and we analyzed 512 links and achieved an accuracy of 98%.

Table 2: Accuracy of CLDIFF

Project	Concise Code Differences		Links	
	Size	Accuracy	Size	Accuracy
RxJava	99	1.00	26	1.00
elasticsearch	88	1.00	24	1.00
okhttp	88	0.98	52	0.85
retrofit	78	1.00	31	1.00
spring-framework	175	1.00	69	0.99
zxing	83	1.00	36	1.00
netty	122	0.98	42	1.00
fastjson	95	0.98	37	1.00
guava	167	0.99	54	1.00
glide	154	0.99	45	1.00
mybatis-3	129	1.00	38	1.00
MPAndroidChart	178	0.98	58	1.00

For all the 12 inaccurate code differences, we found that all of them were caused by the inaccurate mapping in GUMTREE (because CLDIFF uses the mapping that is heuristically generated by GUMTREE). In detail, 10 of them were caused by missed mappings, i.e. two AST nodes that should have been mapped are actually not mapped. As a result, GUMTREE generates a *delete* and an *add* action instead of a *move* action, making CLDIFF fail to generate a *move* action as well. In addition, two of them were caused by wrong mappings, i.e. two AST nodes that should not have been mapped are actually mapped. Thus, both GUMTREE and CLDIFF generate a code difference that does not reflect the real code change, confusing the change understanding.

Among the 512 links, our five pre-defined links all occurred except for *Implement-Method* links; and around 91% of them were *Def-Use* links. We found totally 9 inaccurate links and all of them were *Def-Use* links. They were caused by our heuristic nature of establishing links; e.g. when a local variable shares the same name as a field in its enclosing class, our approach might construct wrong links. This high accuracy is surprising but still reasonable as code changes are often focused and our simple strategy only analyzes those changed code that contain small sources of inaccuracy.

Summary. Based on the results in Table 2, we can positively answer **RQ1** that CLDIFF had a high accuracy of 99% and 98% for the generated concise code differences and established links.

5.3 Conciseness Evaluation (RQ2)

To analyze whether CLDIFF generates concise (or short) code differences compared to GUMTREE, we measured the length of the edit script (i.e. the number of actions in the script) for each commit. Since the *update* actions in CLDIFF simply put a set of fine-grained actions together but not represent a complete action like our *add* and *delete* actions do, we used the number of those fine-grained actions for the counting for our *update* actions to have a fair comparison. Overall, for 90% commits, CLDIFF generated shorter edit scripts than GUMTREE. For the remaining 10% commits, CLDIFF had the same length as GUMTREE, meaning that the fine-grained edit actions cannot be grouped at or above the statement level.

Table 3 presents the maximum and median length for each project (the minimum lengths are omitted as they are all one), which shows that CLDIFF significantly shortened the edit script. Fig. 6 further shows the length ratio of CLDIFF to GUMTREE with respect to each commit in each project. For all the projects, the median ratio was around 0.2. Numerically, for 48% commits, CLDIFF shortened edit scripts by more than 80%. This owes to our high-level *add* and *delete* actions, describing a group of fine-grained *add* and *delete* actions.

Table 3: Length of Generated Code Differences

Project	Maximum		Median	
	GumTree	ClDiff	GumTree	ClDiff
RxJava	56905	4727	107.5	10
elasticsearch	317867	9695	62	11
okhttp	17325	1039	79	14
retrofit	4738	360	51	8
spring-framework	102587	5972	46	9
zxing	14580	915	36	8
netty	48401	6411	38	8
fastjson	69996	1889	54	8
guava	23820	4276	46	4
glide	23592	902	59	10
mybatis-3	9592	336	31	7
MPAndroidChart	18123	2920	100	21
Average	58961	3287	59	10

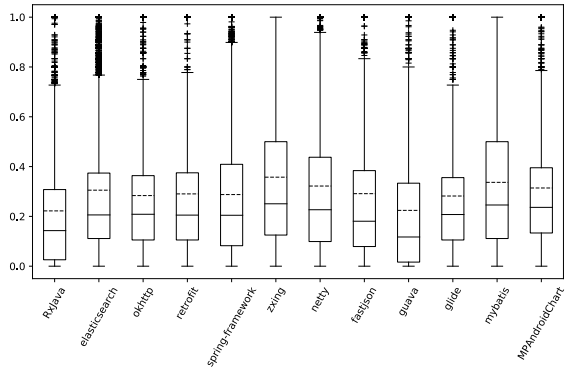


Figure 6: Length Ratio of ClDiff to GumTree

Table 4 lists the maximum and median group size for our *add* and *delete* actions. These maximum cases often correspond to the addition or deletion of an entire method declaration. The median size was respectively 8 and 6 for our *add* and *delete* actions.

Summary. Based on the results in Table 3 and 4 and Fig. 6, we can positively answer **RQ2** that ClDiff generated more than 80% shorter edit scripts for 48% commits than GumTree.

5.4 Performance Evaluation (RQ3)

Table 5 compares the average performance overhead (in milliseconds) of ClDiff and GumTree in generating code differences for the set of changed source code files in each commit. It also reports the performance overhead of each step in ClDiff. We can see that ClDiff took 72% shorter time than GumTree. The reason is that, in ClDiff, we prune unchanged declarations in the AST pairs before applying GumTree to generate fine-grained code differences, while GumTree directly works on raw ASTs. Besides, the second step of ClDiff is the most expensive step, spending 92% of the time. The third step is the cheapest step, only taking 0.42 milliseconds for a commit. This actually owes to our heuristic-based strategy to build links, which also achieves high accuracy as discussed in Section 5.2. On average, ClDiff spent 188.51 milliseconds for a commit.

Summary. Based on the results in Table 5, we can positively answer **RQ3** that ClDiff spent 72% shorter time than GumTree.

5.5 Usefulness Evaluation (RQ4)

To evaluate the usefulness of ClDiff, we conducted a human study with 10 participants to understand the changes in 10 commits (i.e. to finish 10 tasks) with the help of ClDiff and GumTree. This study was

Table 4: Group Size of Our *add* and *delete* Actions

Project	Maximum		Median	
	add	*delete*	*add*	*delete*
RxJava	2540	341	8	6
elasticsearch	2832	2029	8	6
okhttp	851	179	7	5
retrofit	2631	130	7	6
spring-framework	969	1425	8	6
zxing	2797	157	7	5
netty	1955	881	7	5
fastjson	1898	511	7	6
guava	6276	6240	10	6
glide	408	213	8	6
mybatis-3	781	188	8	5
MPAndroidChart	1486	872	7	7
Average	2119	1097	8	6

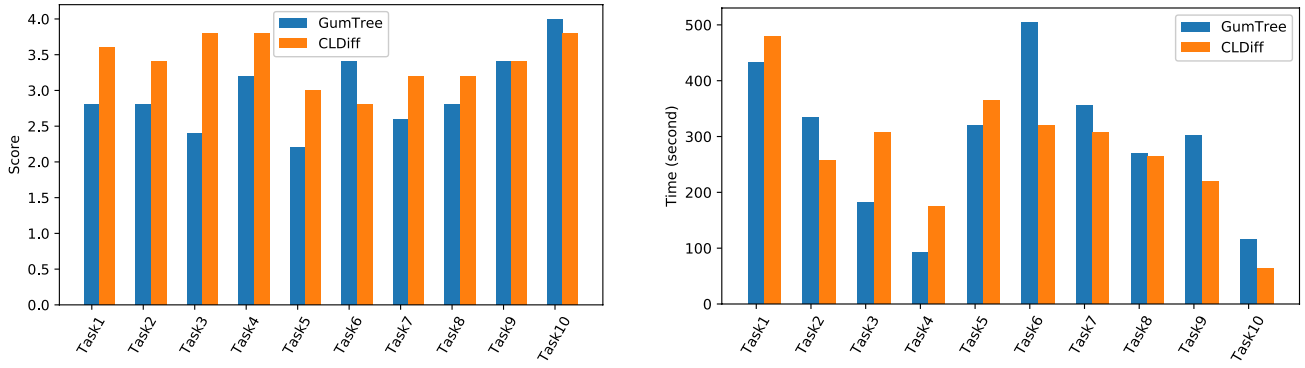
Table 5: Performance Overhead of GumTree and ClDiff

Project	GumTree (ms)	ClDiff (ms)			
		Step 1	Step 2	Step 3	Total
RxJava	1987	38	41	0	79
elasticsearch	869	18	87	3	108
okhttp	379	12	18	0	30
retrofit	185	8	14	0	22
spring-framework	452	9	15	1	25
zxing	221	7	14	0	21
netty	378	9	21	0	30
fastjson	329	16	40	1	57
guava	2306	16	1763	0	1779
glide	244	11	17	0	28
mybatis-3	292	7	9	0	16
MPAndroidChart	416	22	45	0	67
Average	671.50	14.42	173.67	0.42	188.51

conducted blindly; i.e. participants did not know which tool was developed by us. We divided the participants into two groups equally. The first group used ClDiff to understand the changes for the first five tasks and used GumTree for the remaining five tasks. The second group used ClDiff and GumTree in an opposite way. Every participant was asked to answer several questions about the changes in each task, write down a summary of his/her understanding about the changes in each task, and record the time required to finish each task. Details of the 10 tasks are available at [1]. After they finished all the tasks, we further asked the participants to finish a questionnaire which contained four questions with provided options.

- *Q1:* Does ClDiff do a good job?
  (a) Yes, (b) Neutral, (c) No
- *Q2:* Does GumTree do a good job?
  (a) Yes, (b) Neutral, (c) No
- *Q3:* Is ClDiff or GumTree more helpful?
  (a) ClDiff, (b) GumTree, (c) No Difference
- *Q4:* Are ClDiff's code differences and links helpful?
  (a) Both, (b) Code Differences, (c) Links, (d) Neither

Based on this human study, we used three indicators to compare ClDiff with GumTree. The first indicator is a score to assess the degree of understanding the changes in each task. Two of the authors manually assigned a score between 0 and 2 to both the task-specific questions and the summary of each task for each participant. Thus a full score is 4. As task-specific questions had deterministic answers, 0.5 was deducted for one wrong answer. The summary was scored based on whether code changes were understood. Due to the subjective nature, the two authors finalized the summary's score through discussion. The second indicator is the time required to finish each task. The third is the qualitative results about the questionnaire.



(a) Score of Understanding the Changes in 10 Tasks **(b) Time of Understanding the Changes in 10 Tasks**

Figure 7: Comparison Results of the Score and Time of Understanding the Changes in 10 Tasks

Table 6: Answers to the Questionnaire

Question	Answers			
	(a)	(b)	(c)	(d)
Q1	10	0	0	–
Q2	3	4	3	–
Q3	10	0	0	–
Q4	5	2	3	0

Fig. 7 shows the results of the two indicators score and time. The x-axis in Fig. 7 denotes each task, the y-axis in Fig. 7a and 7b respectively denote the average score of understanding the changes in each task and the average time to finish each task. Overall, the average total score of CLDIFF and GUMTREE for 10 tasks was respectively 34.0 and 29.6; and there was a significant difference in score between CLDIFF and GUMTREE according to the Wilcoxon Signed-Rank Test. The average total time of CLDIFF and GUMTREE for 10 tasks was respectively 1,539 and 1,865 seconds. However, there was no significant difference in time between CLDIFF and GUMTREE. Specifically, in four tasks, CLDIFF took more time but had higher score; in two tasks, CLDIFF took less time but had lower score; and in four tasks, CLDIFF took less time and had higher score.

Table 6 reports the results of the four questions in the questionnaire. The first column lists the question, and the other four columns report the number of participants choosing the corresponding options. Generally, all the participant felt that CLDIFF was helpful (*Q1*), and was more helpful than GUMTREE (*Q3*), while some participants felt that GUMTREE was not very helpful (*Q2*). Besides, seven participants thought that our concise code differences were helpful, and eight participants thought that our links were helpful (*Q4*).

Summary. Based on the results in Fig. 7 and Table 6, we can positively answer **RQ4** that CLDIFF was more useful than GUMTREE in understanding code changes for all participants; and our concise code differences and their links were helpful for most participants.

5.6 Discussion

Threats. The primary threats to the validity of our experiments and human study are twofold. First, we analyzed the accuracy of CLDIFF using a total number of 120 commits, which was not very large-scale. This is because such a manual analysis is very time-consuming, involving the understanding of mapping, edit script, AST pairs and real code changes. Hence, we followed the similar work in the literature [24] to use 120 commits. However, these commits were taken from 12 different projects, and thus can be considered as representative code changes. Second, we hired 10 graduate students to participate the human study rather than developers working in the industry. Therefore, we only recruited the students that had at least 3-years programming experience. A further human study is required to evaluate the usefulness of CLDIFF in the industry.

Limitations. One main limitation of CLDIFF is the heuristic nature of establishing links, especially for *Def-Use* links, as indicated in our accuracy evaluation (Section 5.2). We plan to investigate the cost and benefit of using data-flow analysis to further improve the link accuracy. On the other hand, we only support five kinds of links. We plan to further analyze the usefulness of each kind of links, extend the capability of current links and support more links such that we can have a compact but really useful set of links.

Applications. We believe that CLDIFF can be useful in various applications. For example, by applying CLDIFF to the evolution history of a project and chaining these code differences together, we can detect logical coupling [57] at a finer granularity. Using statistics about the different kinds of code differences in each commit as features, we can classify commits [10] into bug fixing, refactoring or upgrading based on machine learning techniques. By further attaching a semantic understanding of our generated code differences, we can characterize or even quantify semantic changes for security patch or compatibility analysis [54, 56]. By combining CLDIFF with performance analysis techniques [7, 12, 13], we can analyze performance regressions and potentially locate their root causes.

6 RELATED WORK

We focus our discussion on the most relevant work in four aspects, i.e. code differencing, code change summarization, code change decomposition, and systematic code changes.

6.1 Code Differencing

Text-based approaches [44, 46] are first proposed to compute differences (in the form of inserted, removed or changed lines of code) between two versions of a source file, followed by several advances [4, 9, 50] that further identify moved lines of code. These approaches are often fast and language-independent; however, they fail to compute syntactic code changes [39], hindering code review, automatic analysis and tool development based on their code differences.

Tree-based approaches [17, 19, 21] are then proposed to generate syntactic code changes. CHANGEDISTILLER [19] uses a general tree

differencing algorithm [11] to generate an edit script from two coarse-grained ASTs where the leaf nodes are code statements (e.g., method invocations or control statements) rather than raw ASTs. Although being sufficient to meet its purpose of classifying certain change types [18], ChangeDistiller is not able to distinguish updates on statements. This also explains why we used and compared GumTree but not ChangeDistiller. Diff/TS [21] can work on raw ASTs. It extends a tree differencing algorithm [58] to generate a fine-grained edit script. A more recent approach is GumTree [17], which also works on raw ASTs. The goal is to find an edit script that well reflects the developer intent based on several heuristics. Higo et al. [24] extend GumTree by introducing copy-and-paste as a new kind of edit actions to make edit scripts shorter and more easily understandable. Dotzler and Philippsen [16] propose some general optimizations to improve the accuracy of the previous tree-based approaches in detecting moved code. Most of these tree-based approaches generate low-level fine-grained representations of code changes, whereas our approach first computes high-level abstracted code changes and then establishes potential links among code changes.

Besides, graph-based differencing approaches [3, 25, 48, 55] are proposed to deal with graph representations of source code, e.g., extended control flow graph [3, 25] and abstract syntax tree [48] with program semantics, and class model [55] with UML semantics. With the semantic information, they can capture certain semantic code changes. Further, some advances [26, 36] have been made to achieve semantic differencing based on input-output behaviors. These approaches provide us with a good insight on extending our approach to understand the semantics behind our syntactic code changes.

6.2 Code Change Summarization

To generate natural language descriptions of code changes, a number of advances [8, 14, 27, 37, 38, 45, 49] have been made to summarize code changes. DeltaDoc [8] captures the behavioral changes for every method and the conditions under which they occur. ChangeScribe [14, 37] generates a commit message by providing a general description of a commit and detailed descriptions of code changes in the commit based on predefined rules. Jiang et al. [27] and Loyola et al. [38] adapt a neural encoder-decoder architecture to automatically generate commit messages from code differences. As software documents are often related, Rastkar and Murphy [49] propose a machine learning-based technique to extract descriptions from a set of relevant documents (e.g., commit messages or bug reports). Integrating the ideas of [37] and [49], ARENA [45] summarizes code changes at the system level and links to issues to generate release notes. These change summarization techniques are mostly designed for the ease of documentation of code changes, while ClDiff generates more fine-grained code changes at the syntactic level.

6.3 Code Change Decomposition

Developers usually commit unrelated or loosely related code changes in a single commit, resulting in tangled changes which make code review difficult and commit-oriented analysis biased. To this end, Kawrykow and Robillard [30] detect non-essential changes (e.g., local variable extractions) in a commit based on fine-grained code change analysis. Herzig and Zeller [23] report the first empirical study on the frequency and impact of tangled changes. They use a multilevel graph-partition algorithm [29] to decompose tangled changes based on a set of features. Dias et al. [15] improve features in [23] by not relying on static analysis but considering fine-grained code change information gathered during development. Based on improved features, they leverage machine learning and clustering to decompose tangled changes. Barnett et al. [5] use *def-use* information from added or changed code to decompose tangled changes. Tao and Kim [53] develop three heuristics to decompose tangled changes into changes for formatting, changes with static dependencies, and changes with similar patterns. Guo and Song [20] apply program slicing and AST searching to interactively decompose tangled code changes for code review and regression testing. These approaches inspire us to explicitly establish links among code changes.

6.4 Systematic Code Changes

Systematic code changes (i.e., similar, related code changes) can be caused by crosscutting concerns [31], API evolution [22, 28], recurring bug fixes [47] or refactoring [35]. Kim et al. [33] first identify such systematic code changes at the method signature level and represent them as logic rules. Then, Kim et al. [32, 34] extend [33] to describe changes within a method body and at a field level. Recently, Zhang et al. [59] propose an interactive approach to allow developers to customize a generated change template and to match the template to summarize systematic changes and locate potential inconsistent or missing changes. Given a systematic code change, McIntyre and Walker [40] discover locations where this change should be applied (if any exist); and Meng et al. [41, 42] further automatically apply this change to the discovered locations with different contexts. Different from these approaches that focus on a specific kind of code changes (i.e. systematic code changes), our approach focuses on a broader range of code changes. Further, we plan to use them to improve the construction of *Systematic-Change* links.

7 CONCLUSIONS

In this paper, we have proposed and implemented a code differencing approach, named ClDiff, to generate concise linked code differences. ClDiff's goal is to generate more easily understandable code differences. Taking as inputs a set of source code files before and after changes, ClDiff works in three steps. First, it pre-processes source code files to prune unchanged declarations from parsed abstract syntax trees. Second, it groups fine-grained code differences at or above the statement level and generates a concise code difference to capture high-level changes in each group. Third, it links the related concise code differences based on five pre-defined links. Our experiments with 12 open-source Java projects and a human study with 10 participants have demonstrated the accuracy, conciseness, performance and usefulness of ClDiff.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000801 and the Shanghai Science and Technology Development Funds (16JC1400801), and was partially supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2016NCR-NCR002-026) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] 2018. CLDIFF. https://github.com/FudanSELab/CLDIFF.

[2] George W Adamson and Jillian Boreham. 1974. The use of an association measure based on character structure to identify semantically related pairs of words and document titles. *Information storage and retrieval* 10, 7-8 (1974), 253–260.

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2004. A Differencing Algorithm for Object-Oriented Programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. 2–13.

[4] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Massimiliano Di Penta. 2013. LHDiff: A language-independent hybrid approach for tracking source code lines. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*. 230–239.

[5] Mike Barnett, Christian Bird, João 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*. 134–144.

[6] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2017. On the Optimal Order of Reading Source Code Changes for Review. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*. 329–340.

[7] Marc Brünink and David S. Rosenblum. 2016. Mining Performance Specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 39–49.

[8] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 33–42.

[9] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2009. Tracking your changes: A language-independent approach. *IEEE Software* 26, 1 (2009), 50–57.

[10] Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González. 2017. GitcProc: a tool for processing and classifying GitHub commits. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 396–399.

[11] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. 493–504.

[12] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating Performance Distributions via Probabilistic Symbolic Execution. In *Proceedings of the 38th International Conference on Software Engineering*. 49–60.

[13] Zhifei Chen, Bihuan Chen, Lu Xiao, Xiao Wang, Lin Chen, Yang Liu, and Baowen Xu. 2018. Speedoo: Prioritizing Performance Optimization Opportunities. In *Proceedings of the 40th International Conference on Software Engineering*. 811–821.

[14] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On Automatically Generating Commit Messages via Summarization of Source Code Changes. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 275–284.

[15] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering*. 341–350.

[16] Georg Dotzler and Michael Philippsen. 2016. Move-optimized Source Code Tree Differencing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 660–671.

[17] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 313–324.

[18] Beat Fluri and Harald C. Gall. 2006. Classifying Change Types for Qualifying Change Couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*. 35–45.

[19] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering* 33, 11 (2007), 725–743.

[20] Bo Guo and Myoungkyu Song. 2017. Interactively Decomposing Composite Changes to Support Code Review and Regression Testing. In *Proceedings of the IEEE 41st Annual Computer Software and Applications Conference*. 118–127.

[21] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*. 279–288.

[22] Johannes Henkel and Amer Diwan. 2005. CatchUp!: Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering*. 274–283.

[23] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. 121–130.

[24] Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto. 2017. Generating Simpler AST Edit Scripts by Considering Copy-and-paste. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. 532–542.

[25] Susan Horwitz. 1990. Identifying the Semantic and Textual Differences Between Two Versions of a Program. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. 234–245.

[26] Daniel Jackson and David A Ladd. 1994. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *Proceedings of the International Conference on Software Maintenance*. 243–252.

[27] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 135–146.

[28] Puneet Kapur, Brad Cossette, and Robert J. Walker. 2010. Refactoring References for Library Migration. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 726–738.

[29] George Karypis and Vipin Kumar. 1995. Analysis of multilevel graph partitioning. In *Proceedings of the IEEE/ACM Conference on Supercomputing*. 29–29.

[30] David Kawrykow and Martin P. Robillard. 2011. Non-essential Changes in Version Histories. In *Proceedings of the 33rd International Conference on Software Engineering*. 351–360.

[31] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*. 220–242.

[32] Miryung Kim and David Notkin. 2009. Discovering and Representing Systematic Code Changes. In *Proceedings of the 31st International Conference on Software Engineering*. 309–319.

[33] Miryung Kim, David Notkin, and Dan Grossman. 2007. Automatic Inference of Structural Changes for Matching Across Program Versions. In *Proceedings of the 29th International Conference on Software Engineering*. 333–343.

[34] Miryung Kim, David Notkin, Dan Grossman, and Gary Wilson Jr. 2013. Identifying and Summarizing Systematic Code Changes via Rule Inference. *IEEE Transactions on Software Engineering* 39, 1 (2013), 45–62.

[35] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 187–196.

[36] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*. 712–717.

[37] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 709–712.

[38] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 287–292.

[39] Jonathan I Maletic and Michael L Collard. 2004. Supporting source code difference analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*. 210–219.

[40] Mark M McIntyre and Robert J Walker. 2007. Assisting potentially-repetitive small-scale changes via semi-automated heuristic search. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. 497–500.

[41] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 329–342.

[42] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering*. 502–511.

[43] T. Mens. 2002. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462.

[44] Webb Miller and Eugene W Myers. 1985. A file comparison program. *Software: Practice and Experience* 15, 11 (1985), 1025–1040.

[45] 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), 106–127.

[46] Eugene W Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (1986), 251–266.

[47] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2010. Recurring Bug Fixes in Object-oriented Programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering*. 315–324.

[48] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. 2004. Dex: A Semantic-Graph Differencing Tool for Studying Changes

in Large Code Bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*. 188–197.

[49] Sarah Rastkar and Gail C Murphy. 2013. Why did this code change?. In *Proceedings of the 2013 International Conference on Software Engineering*. 1193–1196.

[50] Steven P. Reiss. 2008. Tracking Source Locations. In *Proceedings of the 30th International Conference on Software Engineering*. 11–20.

[51] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on software engineering* 22, 8 (1996), 529–551.

[52] 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*. 51:1–51:11.

[53] Yida Tao and Sunghun Kim. 2015. Partitioning Composite Code Changes to Facilitate Code Review. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. 180–190.

[54] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 226–237.

[55] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. 54–65.

[56] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*. 462–472.

[57] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. 2004. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering* 30, 9 (2004), 574–586.

[58] K. Zhang and D. Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262.

[59] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. 2015. Interactive Code Review for Systematic Changes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. 111–122.