CLDIFF: Generating Concise Linked Code Differences

Kaifeng Huang∗
Fudan University
China

Bihuan Chen†
Fudan University
China

Xin Peng∗
Fudan University
China

Daihong Zhou∗
Fudan University
China

Ying Wang∗
Fudan University
China

Yang Liu
Nanyang Technological University
Singapore

Wenyun Zhao∗
Fudan University
China

ABSTRACT
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 message for code changes in a commit [27, 37, 38], and the release

KEYWORDS
Code Differencing, Program Comprehension, AST

CCS CONCEPTS
- Software and its engineering → Software maintenance tools;

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE ’18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09...$15.00
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 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 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 n5 has three child nodes n6, n7 and n8, and its label is MethodInvocation. The label of n6, n7 and n8 is SimpleName. n6, n7 and n8 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. ASTb and ASTa). AST differencing tools can generate an edit script (i.e. a sequence of edit actions). By sequentially applying the
where a newly-declared method (Line 5) is invoked. This new method
(Line 6), a variable is extracted (Line 7–8), and both of them are used

- if
- 29–30) because
- 2
- 2
- 10
- add(n, p, 1)
- 10
- 10
- 1
- move(n, p, 1) moves node n to be the i-th child node of node p.
- delete(n) removes a leaf node n.
- move(n, p, i) moves node n i 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(n5, n13, 2),
which moves the method invocation rooted at n5 to be the second
child node of a variable declaration fragment rooted at n13.

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 state-
ment), but are scattered across the edit script. Such related but scat-
ered 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 con-
sidered as a whole to resolve a conflict. Therefore, to generate more
easily-understandable code differences for both developers and auto-
manual analysis tools, we try to obtain high-level concise code dif-
ferences 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 mov-
ing 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 overrid-
den 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 com-
mit, 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 or-
der 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 ac-
tions) are often related to high-level AST elements but scattered across
the edit script, we generate high-level concise code differences at or
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 \((f_a, f_b)\), we parse it into an AST pair \((AST_b, AST_a)\), 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 initializer 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 \((AST'_b, AST'_a)\). 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 \((AST'_b, AST'_a)\), we use GumTree [17] to generate the mapping \(M\) and the edit script \(A\) between the two ASTs. Recall that \(M\) maintains the mapped AST node pairs and \(A\) 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\((n, p, i)\) \(\in A\), we generate a concise code difference move\((X(n, p, i))\), where \(X\) is the label of node \(n\) and explicitly reflects the syntactic information, and remove move\((n, p, i)\) from \(A\).

**Example 4.1.** The edit script in Fig. 2(d) contains one move action move\((\text{n}5, \text{n}13, 2)\) that moves a whole method invocation. Thus, ClDiff generates moveMethodInvocation\((\text{n}5, \text{n}13, 2)\).

**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 \(N\) can contain statements or declarations, e.g. IfStatement, TryStatement, MethodDeclaration and TypeDeclaration. We define each node \(n \in N\) that is a non-block statement or a declaration as a composing element, each child node of the node \(n \in N\) which is a block statement as a composing element, and all the other nodes in \(N\) 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 \( \text{AST}_A \) (because \( add \) actions are applied on \( \text{AST}_A \)), 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 \( M \) for \( m \)), we mark \( B \) 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 \( B \) 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 \( B \), and remove \( B \) 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 \( B \) and \( C \), and remove \( B \) and \( C \) from \( \mathcal{A} \). If not, we generate \( addXP(n, p, i) \) and remove \( B \) 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 \( \text{AST}_B \) (as delete actions are applied on \( \text{AST}_B \)) 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 \( C_2 \). 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 \( B \) 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 \( C_1 \). We group it with \( add(n_{20}, n_{19}, 1), add(n_{21}, n_{20}, 1), add(n_{22}, n_{21}, 2) \) and \( add(n_{23}, n_{19}, 2) \) in \( \mathcal{B} \). As \( B \) 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 \( C_1 \). We group it with all the other \( add \) actions in Fig. 3(e) in \( \mathcal{B} \). As \( B \) 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 \( \text{AST}_A \), replace \( m \) with its mapping \( m' \) in \( \text{AST}_B \) using \( M \) 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](image-url)

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 \( \text{AST}_B \) 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 \( add(n_{33}, n_2, 2) \) 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 \( addSimpleName(n_{33}, n_2, 2) \) 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
• **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 might be changed through override in each sub-class 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** \( d_1 \xrightarrow{\text{SC}} d_2 \).

Then, we introduce how to establish these links based on concise code differences \( D_i \) 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 \( d \in D_i \) 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 \( e \in D_j \) 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 \( d \in D_i \) 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 \( e \in D_j \) \( (j \neq i) \) 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 \( d \in D_i \) that is applied on node \( n_d \), we first get each delete, add or move action \( e \in D_j \) \( (e \neq d) \) 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 larger than or equal to 0.8, we build the link \( d \xrightarrow{\text{DU}} e \). 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 before and AST after or only from AST before depending on whether \( n_d \) and \( n_e \) can be respectively mapped in their \( M_i \)), 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].
5.1 Evaluation Setup

To evaluate the effectiveness of ClDiff, we conducted experiments using 12 highly-starred 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 actions, describing a group of fine-grained actions, and we analyzed 512 links and achieved an accuracy of 98%.

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>
<thead>
<tr>
<th>Projects</th>
<th>Creation Date</th>
<th>LOC</th>
<th>Stars</th>
<th>Commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RxJava</td>
<td>2012-03</td>
<td>270.0K</td>
<td>32.6K</td>
<td>4226</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>2010-02</td>
<td>889.2K</td>
<td>30.5K</td>
<td>29929</td>
</tr>
<tr>
<td>okhttp</td>
<td>2011-05</td>
<td>60.0K</td>
<td>26.3K</td>
<td>2754</td>
</tr>
<tr>
<td>retrofit</td>
<td>2010-09</td>
<td>22.4K</td>
<td>27.5K</td>
<td>1090</td>
</tr>
<tr>
<td>spring-framework</td>
<td>2008-07</td>
<td>673.5K</td>
<td>20.7K</td>
<td>12838</td>
</tr>
<tr>
<td>zxing</td>
<td>2007-10</td>
<td>156.0K</td>
<td>18.3K</td>
<td>1793</td>
</tr>
<tr>
<td>netty</td>
<td>2008-08</td>
<td>238.6K</td>
<td>13.7K</td>
<td>11047</td>
</tr>
<tr>
<td>fastjson</td>
<td>2011-07</td>
<td>150.0K</td>
<td>13.3K</td>
<td>2304</td>
</tr>
<tr>
<td>guava</td>
<td>2009-06</td>
<td>342.0K</td>
<td>23.7K</td>
<td>3925</td>
</tr>
<tr>
<td>glide</td>
<td>2012-12</td>
<td>73.4K</td>
<td>21.4K</td>
<td>1745</td>
</tr>
<tr>
<td>mybatis-3</td>
<td>2010-05</td>
<td>96.0K</td>
<td>7.4K</td>
<td>1189</td>
</tr>
<tr>
<td>MPAndroidChart</td>
<td>2014-04</td>
<td>28.7K</td>
<td>21.8K</td>
<td>1517</td>
</tr>
</tbody>
</table>

### 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 0.6. 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

<table>
<thead>
<tr>
<th>Project</th>
<th>GumTree (ms)</th>
<th>ClDiff (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RxJava</td>
<td>56.905</td>
<td>41.27</td>
</tr>
<tr>
<td>elasticSearch</td>
<td>31.787</td>
<td>46.24</td>
</tr>
<tr>
<td>okhttp</td>
<td>17.325</td>
<td>10.39</td>
</tr>
<tr>
<td>retrofit</td>
<td>47.38</td>
<td>36.09</td>
</tr>
<tr>
<td>spring-framework</td>
<td>102.587</td>
<td>59.72</td>
</tr>
<tr>
<td>zxing</td>
<td>145.89</td>
<td>91.75</td>
</tr>
<tr>
<td>netty</td>
<td>48.011</td>
<td>64.11</td>
</tr>
<tr>
<td>fastjson</td>
<td>699.96</td>
<td>18.89</td>
</tr>
<tr>
<td>guava</td>
<td>258.20</td>
<td>42.76</td>
</tr>
<tr>
<td>glide</td>
<td>235.92</td>
<td>90.82</td>
</tr>
<tr>
<td>mybatis-3</td>
<td>959.24</td>
<td>33.64</td>
</tr>
<tr>
<td>MPAndroidChart</td>
<td>181.23</td>
<td>292.00</td>
</tr>
<tr>
<td>Average</td>
<td>588.61</td>
<td>328.71</td>
</tr>
</tbody>
</table>

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 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.
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 Philippens [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.


