Pruning the AST with Hunks to Speed up Tree Differencing

Chunhua Yang
School of Computer Science and Technology
QILU University of Technology(Shandong Academy of Sciences)
Shandong, China
jnych@126.com

E. James Whitehead
Dept. of Computational Media
University of California, Santa Cruz
Santa Cruz, USA
ejw@soe.ucsc.edu

***Abstract*—Inefficiency is a problem in tree-differencing approaches. As textual-differencing approaches are highly efficient and the hunks returned by them reflect the line range of the modified code, we propose a novel approach to prune the AST with hunks. We define the pruning strategies at the declaration level and the statement level, respectively. And, we have designed an algorithm to implement the strategies. Furthermore, we have integrated the algorithm to ChangeDistiller and GumTree. Through an evaluation on four open source projects, the results show that the approach is very effective in reducing the number of nodes and shortening the running time. On average, with declaration-level pruning, the number of nodes in the two tools is reduced by at least 64%. With statement-level pruning, the number of nodes in both tools is reduced by at least 74%. By using the declaration-level pruning and the statement-level pruning, GumTree's runtime is reduced by at least 70% and 75%, respectively.**

***Index Terms*—tree-differencing, tree pruning, software evolution, program understanding**

I. Introduction

Differencing tools play an important role in understanding how software changes and evolves. With the change set provided by these tools, users can know where the code has changed without having to check the entire source code.

Textual-differencing tools such as GNU diff [1] and Ldiff [2] provide a set of hunks containing code lines that have been removed or added. They are highly efficient and therefore used prevalently in many version management systems for presenting a difference view. However, hunks do not carry syntax information, which makes it difficult for deep analysis.

Tree-differencing tools such as ChangeDistiller [3], GumTree [4], and Diff/TS [5] generate a set of changes by matching and differencing two versions of the AST of the source code. Changes returned by these tools carry syntax information, which have been utilized to understand software evolution, including predicting types of code changes [6], identifying components evolving together [7], analyzing the impact of changes [8], grouping the changes involved in the same task [9], etc.

However, one of the main disadvantages of tree-differencing tools is their inefficiency compared to textual-differencing tools. This is especially apparent when dealing with large-scale code files.

In general, a tree-differencing approach consists of two phases: a matching phase and a differencing phase. In the first phase, two versions of the AST of the source code are matched each other to find mappings of nodes. In the second phase, an edit script is generated by comparing the nodes in each mapping. The time complexity of a tree-differencing approach is $O(n^2)$, in which n is the number of nodes in the tree used for matching. Therefore, if we cut down the number of nodes in the matching tree, then the average runtime would be reduced.

As textual-differencing tools are fast and the hunks returned by them reflect the line range of the modified code, we propose to prune the tree in tree-differencing tools using hunks. We propose strategies for pruning declarations and statements. And we have designed an algorithm to implement the strategies. The algorithm generates a filter tree for the old version and new version of the AST, respectively. The filter tree is then used for trimming nodes in the process of generating the matching tree.

The remainder of the paper is organized as follows. In Section 2, we introduce a motivation example, outline of the approach, and the pruning strategies. In Section 3, we present the pruning algorithm. In Section 4, we describe the integration of the algorithm to ChangeDistiller and GumTree. We present the evaluation in Section 5. In Section 6, we review the related work. We conclude the paper in Section 7.

II. The Approach

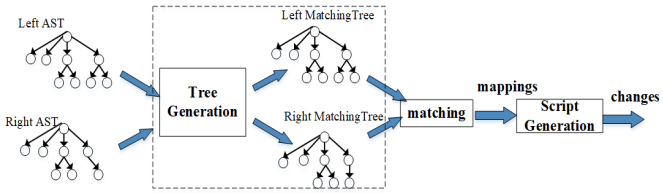
In this section, we outline the approach, introduce an example to illustrate the pruning, and present the pruning strategies.

A. Outline of the Approach

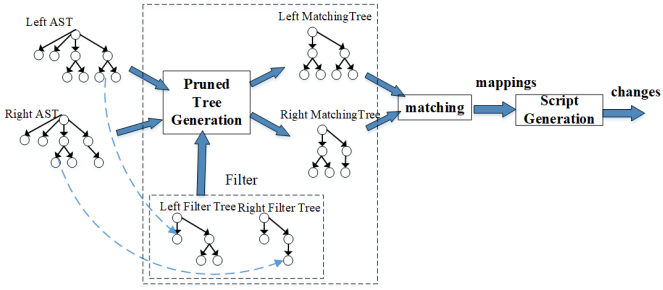
Generally, tree-differencing approaches extract changes by comparing the old version and new version of the AST of the source code. The process is shown in Fig.1(a). They usually first create their own tree structure by traversing the AST. We call the process *tree generation*. Then, two versions of the tree input to the *matching* step to calculate the mappings between their similar nodes. Based on the mappings, in the next step *script generation*, the edit script will be deduced. An edit script is a sequence of actions, usually called $changes$.

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

We propose to prune the nodes in the matching tree. It intervenes the process of tree generation. The outline is illustrated in Fig.1(b). Firstly, we build two versions of *filter tree*, based on the pruning strategies proposed in the next section. Then, during tree generation, an AST node will add to the matching tree only if it is in the filter tree.



(a) The original tree-differencing



(b) The pruned tree-differencing

Fig. 1. Original tree-differencing and the pruned tree-differencing.

B. An Example for Illustrating the Pruning

Fig.2 illustrates the pruning. Two hunks are listed in the lower part of the figure. They are extracted by GNU diff from two versions of file *SourceProvider.java* in revision *d9e0545* of project Guice[1].

A hunk consists of removed or added lines. The removed lines are prefixed with a minus sign and colored in red, while the added lines have a plus prefix and are colored in green. The lines before and after the hunk are contextual and remain unchanged.

The first hunk in the figure shows that the field $parent$ has been added to the new version. The second hunk tells us that the assignment $this.classNamesToSkip = ImmutableSet.copyOf(classesToSkip)$ has been removed from the old version, while the method invocation $this(null, classesToSkip)$ and the constructor $SourceProvider(SourceProvider, Iterable < String >)$ have been added to the new version.

Both versions of the AST are shown above the two hunks. Take the tree on the left for illustration. The root is a compilation unit *LeftCU*. It has a type declaration, i.e. class $SourceProvider$, which contains two fields $UNKNOWN\_SOURCE$ and $classNamesToSkip$, and two methods $SourceProvider$ and $asStrings$. According to the hunks in the figure, the two fields remain unchanged. The assignment in method $SourceProvider$ has been removed. The method $asStrings$ does not appear in the hunks and remains unchanged.

We refer to nodes in hunks as *shadow* nodes. For example, the assignment in the left tree is a *shadow* statement. The main principle of pruning is that if a node and its children do not appear in any hunks, then it should be cut. Therefore, in the left tree, the two fields and the method $asStrings$ should be cut. In the right tree, the fields $UNKNOWN\_SOURCE$ and $classNamesToSkip$ and the method $asStrings$ should be cut.

C. The Declaration-Level Pruning

1) The Crossing Relationship Among Hunks and Declarations: If the range of an entity overlaps with the ranges of hunks, then we say that these hunks $cross$ the entity. For example, in Fig.2 the second hunk crosses the assignment node of the left tree.

A hunk can cross an deleted, added, or modified declaration, as illustrated in Fig.3(a)∼(c). Sometimes, a hunk may cross multiple declarations. As shown in Fig.3(d), a hunk crosses method A and B, in which B is a new declaration. This is a method split. The second hunk in Fig.2 is also such a kind.

2) The Coupling Relationship Between Old Version Declarations and New Version Declarations: Suppose A is a declaration of the old AST and B is one of the new AST. If they are the same type and there are hunks crossing them, then we call A and B *coupled* to each other.

The coupling between two declarations takes the following three forms:

- one-one: one declaration in the old version of the AST is coupled to one declaration in the new version of the AST.
- one-more/more-one/more-more: one declaration in the old version of the AST is coupled to multiple declarations in the new version of the AST, or multiple declarations in the old version of the AST are coupled to multiple declarations or one declaration in the new version of the AST.
- zero-one/zero-more/one-zero/more-zero: the first two mean that a hunk crosses one or more newly added declaration(s), while the last two refer to that a hunk crosses one or more removed declaration(s).

3) Pruning Declarations: Based on the coupling relationship between the declarations, we make the following pruning strategies:

- For declarations that follow a zero-one/zero-more/one-zero/more-zero coupling relationship, we keep them all in the filter tree.
- For each declaration involved in an one-more/more-one/more-more coupling, we keep the whole declaration in the filter tree. This means that both its signature and its body members stay in the filter tree, whether or not they appear in hunks.
- For all deleted, added, or updated fields, regardless of their coupling relationship, we keep them all in the filter tree. Since a field has no body, it either remains in the filter tree or is removed from the filter tree.

[1]https://github.com/google/guice/

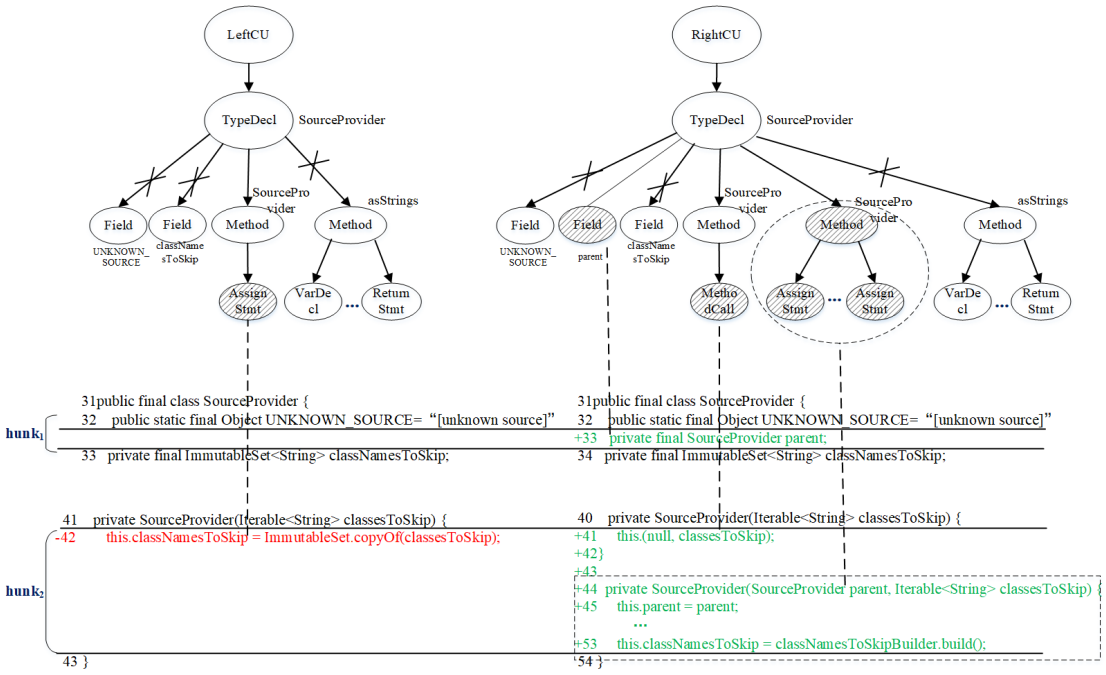


Fig. 2. Illustration of the pruning.

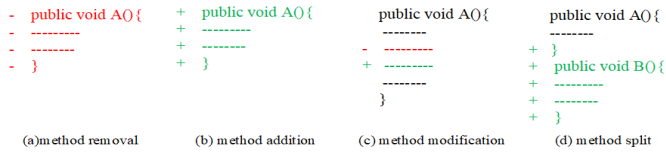


Fig. 3. A sample hunk with method removal, addition, modification, or split.

- In the case of one-one coupling, we use the following strategies for different kinds of declarations:
  - For a pair of coupled classes, we keep each of their signatures in the filter tree. Then, we will examine their body members to find the coupling relation between them and handle them recursively.
  - For a pair of coupled methods, we keep each of their signatures in the filter tree. Then, we will prune their body statements using the strategies to be described in the next subsection.

4) The Statement-Level Pruning: On the statement level, although we can filter out each statement that does not appear in any hunks, we do not, for the following two reasons:

- Sometimes, the order of a statement changes and there are only context lines between the two locations, as illustrated in Fig.4(a). If the contextual statements are pruned, the movement of the statement in the first hunk to the second hunk can not be recognized. Because, at the matching stage, the statement $x = 1$ in the old version tree will match the one in the new version tree.
- For a structure statement, if we filter out all of its inner statements that did not occur in any hunks, then the changes detected from the pruned version may be inconsistent with those from the original version.

For example, in the If-statement as illustrated in Fig. 4(b), only an inner statement was removed. Usually, the original differencing algorithm will identify a statement update on the If-statement. However, if we filter out the unchanged statements, then the pruned algorithm may return two changes, i.e. the removal of the old version If-statement and the addition of the new version If-statement. This is because most current differencing algorithms check a statement update based on the similarity between the value of the old version and the value of the new version. If only shadow statements are kept, the similarity between the two versions of the If-statement will reduce. If it is below the threshold set by the algorithm, then the two statements are considered to be two different statements.

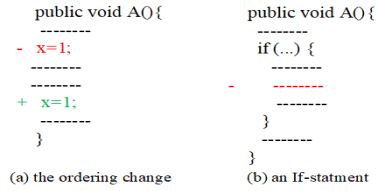


Fig. 4. Example statements

To avoid the two issues, we make the following rules for pruning the statements in a method:

- We keep all the statements between the first shadow statement and the last shadow statement in the filter tree.

- We limit the pruning to the first-level statements, i.e. the statements in the first-level of the method tree. That means, if a statement is a first-level statement and should be saved in the filter tree, then its child statements will also be kept.

Fig.5 shows three hunks that cross the body of a constructor. They are from $HistoryTextField.java$ in revision $e0e7d11$ from $jEdit$[2]. For the old version, we will add all the statements between line #31 and line #40 to the filter tree. For the new version, we will add all the statements between line #31 and line #38 to the filter tree.

```
29 29     public HistoryTextField(String name)
30 30     {
31    -       history = new DefaultComboBoxModel();
32 31         this.name = name;
33 32         String line;
34 33         int i = 0;
35 34         while((line = jEdit.getProperty("history." + name + "." + i)) != null)
36 35         {
37    -           history.addElement(line);
   36 +           addItem(line);
38 37             i++;
39 38         }
40    -       setModel(history);
41 39         setEditable(true);
42 40         setMaximumRowCount(10);
43 41         setSelectedItem(null);
```

Fig. 5. Example statement changes.

III. The Pruning Algorithm

We have designed the algorithm to implement the pruning strategies. It inputs two versions of the AST and a set of hunks that have been extracted from the source code, and then generates two versions of the filter tree.

Each node in a filter tree is composed of an AST node, a flag, and a set of child nodes. The AST node is a declaration or statement that should be kept in the matching tree. The flag indicates whether all the child nodes of the AST node should also be retained. If the flag has a value of 1, then all of its children should be kept. With such a flag, we avoid generating all child nodes for the AST node with a flag 1. This is to minimize the number of nodes in the filter tree. For the example changes shown in Fig.2, two filter trees as shown in Fig.6 will be generated. The number in each node is the flag.

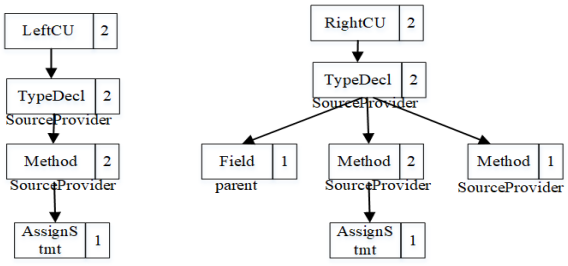


Fig. 6. The filter tree that will be generated for the changes shown in Fig.2.

Starts from the roots, the algorithm builds both versions of the filter trees based on a breadth-first traversal on the two versions of the AST.

[2]https://github.com/linzhp/jEdit-Clone

A. The Main Algorithm FilterCU

Algorithm $FilterCU$ is the main algorithm. It inputs a set H of hunks, compilation unit $leftcu$ and $rightcu$ that are the roots of the old and new version of the AST respectively. It outputs the root r_1 of the filter tree for $leftcu$, and the root r_2 of the filter tree for $rightcu$.

The algorithm first obtains a set T_1 of shadow declarations from $leftcu$, as well as a set T_2 of shadow declarations from $rightcu$. Then, it invokes algorithm $FilterTypes$ to prune the two sets of shadow declarations, which returns a set R_1 of filter tree nodes for T_1, as well as a set R_2 of filter tree nodes for T_2. Finally, the algorithm creates a node r_1 as the root of the old version filter tree, and it adds all nodes of R_1 as its children. Similarly, it creates the root r_2 of the new version filter tree, and adds all nodes of R_2 as its children.

Algorithm 1: FilterCU($H, leftcu, rightcu$)

Input: a set H of hunks, compilation unit $leftcu$ and $rightcu$

Output: root r_1 of the old version filter tree, root r_2 of the new version filter tree

1 $T_1 \leftarrow getShadowDecls(H, leftcu.types)$;
2 $T_2 \leftarrow getShadowDecls(H, rightcu.types)$;
3 $(R_1, R_2) \leftarrow FilterTypes(H, T_1, T_2)$;
4 $r1 \leftarrow newNode(leftcu, 2), r_2 \leftarrow newNode(rightcu, 2)$;
5 **if** $R_1 \neq \varnothing$ **then** $r_1.addChildren(R_1)$;
6 **if** $R_2 \neq \varnothing$ **then** $r_2.addChildren(R_2)$;

B. Algorithm FilterTypes

Algorithm $FilterTypes$ first calls the function $getCoupledDecls$ to return coupled type declarations.

Function $getCoupledDecls(T_1, T_2)$ returns a set M of tuples. Each tuple $m \in M$ consists of coupled declarations and hunks that cross these declarations. The form of each tuple is as follows:
$m = (\{d_1, \cdots, d_x\}, \{e_1, \cdots, e_y\}, H)$, $d_i \in T_1 \wedge e_j \in T_2 \wedge 1 \leq i \leq x \wedge 1 \leq j \leq y \wedge 0 \leq x \leq |T_1| \wedge 0 \leq y \leq |T_2|$. If x is 0, then the left part of the tuple is empty. If y is 0, then the right part of the tuple is empty.

Then, the algorithm calls function $makeNodes$ to generate the filter tree node for each coupled declaration in m. The flag of each filter node is set to 1 by default. Then, it adds all the nodes to the result sets.

Next, if the coupled declarations in tuple m is not a one-one type, then the algorithm ends. Otherwise, it prunes the body declarations of the coupled type declarations, i.e. line 7∼14. The process is as follows:

- It firstly gets the filter nodes $node_1$ and $node_2$ from S_1 and S_2 respectively, and sets their flag as 2.
- Then, it uses function $getElements$ to parse the elements from tuple m, including the type declaration d_1 and d_2 in the old and new version, and the set H_{sub} of hunks crossing the two declarations.

- Next, it calls $filterBodyDecls$ to prune the body declarations of d_1 and d_2, which returns two sets CR_1 and CR_2 of the filter nodes. All the nodes in CR_1 and CR_2 are added as child nodes of d_1 and d_2 respectively.

Algorithm 2: FilterTypes(H, T_1, T_2)

Input: a set H of hunks, a set T_1 of type declarations in the old version AST, a set $T2$ of type declarations in the new version the AST

Output: two sets C_1 and C_2 of filter tree nodes

$C_1 \leftarrow \varnothing$; $C_2 \leftarrow \varnothing$;
$M \leftarrow getCoupledDecls(T_1, T_2)$;
foreach $m \in M$ **do**
  $(S_1, S_2) \leftarrow makeNodes(m)$;
  if $S_1 \neq \varnothing$ **then** $C_1 \leftarrow C_1 \cup S_1$;
  if $S_2 \neq \varnothing$ **then** $C_2 \leftarrow C_2 \cup S_2$;
  if $isOneOneType(m)$ **then**
    $node_1 \leftarrow s_1, s_1 \in S_1$;
    $node_2 \leftarrow s_2, s_2 \in S_2$;
    $node_1.flag \leftarrow 2, node_2.flag \leftarrow 2$;
    $(d_1, d_2, H_{sub}) \leftarrow getElements(m)$;
    $(CR_1, CR_2) \leftarrow filterBodyDecls(d_1, d_2, H_{sub})$;
    if $CR_1 \neq \varnothing$ **then** $node_1.addChildren(CR_1)$;
    if $CR_2 \neq \varnothing$ **then** $node_2.addChildren(CR_2)$;

C. Algorithm FilterBodyDecls

Algorithm $FilterBodyDecls$ prunes the body declarations of one-one coupled type declarations t_1 and t_2. It consists of three parts.

In the first part, it processes field declarations, by creating a new node for each shadow field and adds it to the result set.

In the second part, it processes method declarations. The process is similar to pruning type declarations. However, when a tuple of one-one coupled methods encountered, only if the granularity is the statement-level, i.e. with a value "$stmt$", it will call $filterBodyStmts$ to process the body statements.

In the third part, it calls the $FilterTypes$ recursively to process the member type declarations.

D. Algorithm FilterBodyStmts

Algorithm $FilterBodyStmts$ prunes statements in one-one coupled methods t_1 and t_2. It first uses $calculateStmtRangeInHunks$ to get the range between the first and the last shadow statement. Then, it uses $getStmts$ to get the statements within the range. Finally, for each statement, it creates a node and adds it to the result set.

E. The Time Complexity and Discussion

The function $getShadowDecls(H, decls)$ has a time of $O(|H| \times |decls|)$. The function $getCoupledDecls(H, D_1, D_2)$ has a time of $O(|H| \times |D_1| \times |D_2|)$.

The time complexity of algorithm $FilterBodyStmts(H, t_1, t_2)$ is $O(|H| \times max(RangeStmts(t_1), RangeStmts(t_2)))$ in which

Algorithm 3: FilterBodyDecls(H, t_1, t_2)

Input: a set H of hunks, a type declaration t_1 in the old version, a type declaration t_2 in the new version

Output: two sets C_1 and C_2 of filter tree nodes

$C_1 \leftarrow \varnothing$; $C_2 \leftarrow \varnothing$;
$Fields_1 \leftarrow getShadowDecls(H, t_1.fields)$;
$Fields_2 \leftarrow getShadowDecls(H, t_2.fields)$;
foreach $field \in Fields_1$ **do**
  $C_1 \leftarrow C_1 \cup \{newNode(field, 1)\}$;
foreach $field \in Fields_2$ **do**
  $C_2 \leftarrow C_2 \cup \{newNode(field, 1)\}$;
$Methods_1 \leftarrow getShadowDecls(H, t_1.methods)$;
$Methods_2 \leftarrow getShadowDecls(H, t_2.methods)$;
$M \leftarrow getCoupledDecls(Methods1, Methods2)$;
foreach $m \in M$ **do**
  $(S_1, S_2) \leftarrow makeNodes(m)$;
  if $S_1 \neq \varnothing$ **then** $C_1 \leftarrow C_1 \cup S_1$;
  if $S_2 \neq \varnothing$ **then** $C_2 \leftarrow C_2 \cup S_2$;
  if $granularity =$ "$stmt$" $\wedge\, isOneOneType(m)$ **then**
    $node_1 \leftarrow s_1, s_1 \in S_1$;
    $node_2 \leftarrow s_2, s_2 \in S_2$;
    $node_1.flag \leftarrow 2, node_2.flag \leftarrow 2$;
    $(d_1, d_2, H_{sub}) \leftarrow getElements(m)$;
    $(CR_1, CR_2) \leftarrow filterBodyStmts(d_1, d_2, H_{sub})$;
    if $CR_1 \neq \varnothing$ **then** $node_1.addChildren(CR_1)$;
    if $CR_2 \neq \varnothing$ **then** $node_2.addChildren(CR_2)$;
$Types_1 \leftarrow getShadowDecls(H, t_1.membertypes)$;
$Types_2 \leftarrow getShadowDecls(H, t_2.membertypes)$;
$(TCR_1, TCR_2) \leftarrow FilterTypes(H, Types_1, Types_2)$;
if $TCR_1 \neq \varnothing$ **then** $C_1 \leftarrow C_1 \cup TCR_1$;
if $TCR_2 \neq \varnothing$ **then** $C_2 \leftarrow C_2 \cup TCR_2$;

Algorithm 4: FilterBodyStmts(H, t_1, t_2)

Input: a set H of hunks, a method declaration t_1 in the old version, a method declaration t_2 in the new version

Output: two sets C_1 and C_2 of filter tree nodes

$C_1 \leftarrow \varnothing$; $C_2 \leftarrow \varnothing$;
$range_1 \leftarrow calculateStmtRangeInHunks(t_1, H)$;
$range_2 \leftarrow calculateStmtRangeInHunks(t_2, H)$;
$Stmts_1 \leftarrow getStmts(range_1, t_1.statements)$;
$Stmts_2 \leftarrow getStmts(range_2, t_2.statements)$;
foreach $stmt \in Stmts_1$ **do**
  $C_1 \leftarrow C_1 \cup \{newNode(stmt, 1)\}$;
foreach $stmt \in Stmts_2$ **do**
  $C_2 \leftarrow C_2 \cup \{newNode(stmt, 1)\}$;

$RangeStmts$ represents the number of statements in the range of the declaration. In the worst case, the time complexity is $O(|H| \times max(Stmts(t_1), Stmts(t_2)))$, in which $Stmts$ is the number of statements in the declaration.

For the main algorithm, the worst case is that all declarations in both versions of the AST should be examined to the statement-level. Let $decls_1$ and $decls_2$ be the number of declarations in the old and new versions of the AST, respectively. Let $stmts_1$ and $stmts_2$ be the maximum number of statements in methods of the old and new versions of the AST, respectively. Then, in the worst case, the time complexity of the pruning algorithm is $O(|H| \times |decls_1| \times |decls_2| \times max(stmts_1, stmts_2))$.

In addition, in the algorithm, we assume that hunks are identified by a line-based approach such as GNU diff. Therefore, the source code format also affect the trimming effect. From a code format perspective, ideally, the pruning works best when each token is placed on its own line. At the other extreme, if the entire program is on one line, then the entire program would not be pruned.

IV. Tuning the Generation of Matching Tree

Once the filter tree is created, it can be integrated into the generation of the matching tree to filter out nodes that should not remain in the tree. Generally, generation of a matching tree is based on the visitors provide by a parser. For example, the eclipse jdt parser provides an $ASTVisitor$ to traverse each node in the AST.

Tree-differencing tools such as ChangeDistiller and GumTree extend the visitor to build their tree for matching. Each node of the tree contains an AST node and other information. Therefore, we tune the generation of matching tree by adjusting the extended visitor in a tree-differencing tool. The process is as follows:

- Before visiting an AST node, a filter function is called to check if it is in the filter tree. If the checking process passes, the AST node will be accessed.
  We use a stack to store the flag of the parent of the node to be accessed. If the flag at the top of the stack is 1, then the node should be visited by default. If the flag is 2, then the filter tree will be checked. If the node is in the filter tree, then it will be accessed. Otherwise, it should not be visited. Before a node is visited, its flag will be pushed to the stack.
- After accessing the AST node, the flag at the top of the stack will pop up.

Pseudo code segments for filtering before and after visiting a node are shown in Algorithm 5 and 6.

We encapsulate the filter tree in a filter container. The container has the root of the filter tree, and a hash table that maps an AST node to a filter node. The hash table guarantees the high efficiency of checking an AST node in the filter. We have integrated the filter container to ChangeDistiller and GumTree for pruning.

A. Application to ChangeDistiller

In the case of ChangeDistiller, the matching tree is generated in two phases. In the first phase, it generates a structure tree that contains only declarations. In the second phase, it generates a body tree for each method that needs to be processed to extract statement-level changes. It provides a $JavaStructureTreeBuilder$ class to build the structure tree, and a $JavaBodyDeclarationConverter$ class to build the method body tree. We tune ChangeDistiller in the following ways:

- We create a new $MyJavaStructureTreeBuilder$ class that extends $JavaStructureTreeBuilder$ to add a filter container.
- We create a new $MyJavaBodyDeclarationConverter$ class that extends $JavaBodyDeclarationConverter$ to add a filter container.

Furthermore, we provide three versions of ChangeDistiller.

- V1- the original version. It uses $JavaStructureTreeBuilder$ as the tree builder and $JavaBodyDeclarationConverter$ as the method body builder.
- V2- the version using the declaration-level pruning. It uses $MyJavaStructureTreeBuilder$ as the tree builder and $JavaBodyDeclarationConverter$ as the method body builder.
- V3- the version using the statement-level pruning. It uses $MyJavaStructureTreeBuilder$ as the tree builder and $MyJavaBodyDeclarationConverter$ as the method body builder.

B. Application to GumTree

GumTree provides a $JdtVisitor$ to generate the matching tree. We adjust GumTree by the following way:

- We create a $MyJdtVisitor$ that extends $JdtVisitor$ to add the filter container. It takes a granularity parameter.

Also, we provide three versions of GumTree.

- V1-the original version. It uses $JdtVisitor$ as the tree builder.
- V2- the version using the declaration-level pruning. It uses $MyJdtVisitor$ with a granularity of 2 as the tree builder.

Algorithm 5: boolean beforecheck(ASTNode node)

```
1 if parentflagstack.peek() == 1 then
2 |   parentflagstack.push(parentflagstack.peek());
3 else
4 |   if ! inFilterTree(node) then return false ;
5 |   parentflagstack.push( this.getFlag(node));
6 return true;
```

Algorithm 6: afterCheck(ASTNode node)

```
1 if parentflagstack.peek() == 1 then
2 |   parentflagstack.pop();
3 else
4 |   if !inFilterTree(node) then return ;
5 |   parentflagstack.pop();
```

- V3- the version using the statement-level pruning. It uses $MyJdtVisitor$ with a granularity of 3 as the tree builder.

C. Limitations

Currently, we only integrate the pruning algorithm into ChangeDistiller and GumTree. They are all written in Java and are based on the eclipse jdt parser. In addition, our current implementation has the following limitations:

1) We did not process comments;
2) We did not handle the default constructor $clinit()$. This causes a problem. When an explicit constructor is modified, the trimmed version of ChangeDistiller returns different changes from the original version. Details will be discussed in the next section.

V. Evaluation

In this section, we present the evaluation. Our aim is twofold:

1) To check the correctness of the pruning algorithm by comparing the changes returned by the pruned version and those by the original version.
2) To check the effectiveness of the approach in reducing nodes and runtime.

The data set is taken from four open projects: jEdit, eclipse JDT Core[3], Apache maven[4], and google-guice. Using a MininGit tool [5], we extracted the change history of the selected projects from their source repository and stored into a database.

Information of the selected projects is shown in Table I. We filtered out revisions containing no modified code files. Also, we filtered out test files. A total of 19,910 remaining revisions were used in the case study. And, we only considered code files from which both ChangeDistiller and GumTree changes identified. As a result, a total of 60,374 files remained. We numbered the four projects as #1, #2, #3, and #4. They are referenced by other tables in the section.

The evaluation was conducted on a MacBook Pro retina with a 2.8 GHz Intel Core i5 with 8 GB of RAM. And, we use a java version of the GNU diff algorithm translated by Stuart D. Gathman[6] to identify hunks from two versions of a java source code file.

A. Reduction in Nodes

We calculated the number of nodes in the matching tree for each file in every commit. The number of nodes per file is the average of the number of nodes in the old and new version of tree. And, we totaled the average number of nodes in all files for each project.

For ChangeDistiller, we define the number of nodes as the number of declarations and statements because it limits granularity to the statement-level. In the case of GumTree, since it is a full-tree differencing tool, we calculated the number of all nodes in the tree, including declarations, statements, expressions, and other types.

[3]https://github.com/eclipse/eclipse.jdt.core
[4]https://github.com/apache/maven/
[5]MiniGit: https://github.com/SoftwareIntrospectionLab/MininGit
[6]http://www.bmsi.com/java/#diff

TABLE I
Study projects, and their total number of revisions containing code files in the time period. Here, $Revisions$ refers to the number of revisions, while $Files$ refers to the number of files.

No.	Project	Period	Revisions	Files
#1	jEdit	1998-09-27∼2012-08-08	4,076	13,830
#2	JDT Core	2001-06-05∼2013-10-16	10,782	33,314
#3	Maven	2003-09-01∼2014-01-29	4,420	10,769
#4	Guice	2006-08-22∼2013-12-11	632	2,461
	Total		19,910	60,374

The average number of nodes in each version per project are listed in Table II. In the case of ChangeDistiller, according to the numbers shown in the table, for jEdit, the average number of nodes in version V2 is 50, while the average number of nodes in version V1 is 396. Therefore, the average number of nodes in version V2 is 12.6% of that in version V1, while the average number of nodes in version V3 is 7.8% of that in version V1. As for JDTCore, Maven, and Guice, the percentage of V2 to V1 is 11.8%, 24.3%, and 19.3%, respectively. The percentage of V3 to V1 is 8.3%, 15.8%, and 13.6%, respectively. Therefore, by using pruning, nodes in ChangeDistiller have greatly reduced.

However, it is not very accurate to calculate the reduction rate by dividing the average number in the original version by that in the pruned version. Because, suppose in version V1 file A has the same number of nodes as the average. In version V2, the number of nodes in file A may not be the same as the average.

Therefore, in order to get a more accurate reduction rate, we calculated the reduction rate V2/V1 in each file, i.e. the average number of nodes in V2 divided by the average number of nodes in V1. We calculated the reduction rate V3/V1 similarly. The average reduction rate for each project is listed in the last two rows in table II. In the case of ChangeDistiller, the average number of nodes in each project is reduced by at least 64% after using the declaration-level pruning, while it is reduced by at least 74% after using the statement-level pruning. In the case of GumTree, the average number of nodes in each project is reduced by at least 66% after using the declaration-level pruning, while it is reduced by at least 75% after using statement-level pruning.

B. Reduction in Runtime

We calculated the runtime of the three versions of ChangeDistiller and Gumtree, respectively. The calculation is through recording the time before the start of the action and the time immediately after the action ends. The time is in milliseconds.

We calculated the following kinds of time when running each tool:

- The time on generating the matching tree: It includes the time spent on generating the old and new versions of the matching tree, and the time on generating the filter trees;

TABLE II
AVERAGE NUMBER OF NODES PER PROJECT WHEN RUNNING THREE VERSIONS OF CHANGEDISTILLER AND GUMTREE RESPECTIVELY. $\#X$ INDICATES THE PROJECT NUMBER, E.G. #1 REFERS TO JEDIT.

Version	In the case of ChangeDistiller				In the case of GumTree			
	#1	#2	#3	#4	#1	#2	#3	#4
V1	396	617	152	88	2,787	4,752	1,266	961
V2	50	73	37	17	381	572	302	199
V3	31	51	24	12	250	407	207	155
V2/V1	0.3	0.22	0.36	0.31	0.31	0.24	0.34	0.32
V3/V1	0.2	0.16	0.26	0.25	0.21	0.18	0.25	0.27

TABLE III
AVERAGE TOTAL TIME PER PROJECT WHEN RUNNING THREE VERSIONS OF CHANGEDISTILLER AND GUMTREE RESPECTIVELY.

Version	In the case of ChangeDistiller				In the case of GumTree			
	#1	#2	#3	#4	#1	#2	#3	#4
V1	6.02	100.9	3.53	1.90	70.87	193	30.57	30.87
V2	5.54	100.8	3.36	1.64	13.31	28.33	9.41	6.56
V3	2.70	85.77	1.93	1.33	10.78	22.16	7.45	6.24
V1-V2	0.48	−1.83	0.17	0.26	57.56	164.6	21.15	24.31
V1-V3	3.32	15.16	1.6	0.57	60.09	170.8	23.12	24.63

- The matching time: the time spent on matching two versions of the tree for getting the mappings;
- The time on script generation: the time on generating the scripts;
- The total time: It is the sum of the other three kinds of time. Note that the total time does not include the time for parsing the AST from the source code.

The mean total time of running each tool on the four projects are listed in Table III. In the table, the first three rows are the total time of running each version of the tool. For ChangeDistiller, the reduction in the mean time is not apparent. For GumTree, the reduction is very obvious.

Also, we computed the reduction time by subtracting the time on running version V2 from the time on running V1 per file. And, we computed the mean of the reduction time in each project. The results are listed in the row starting with *V1-V2*. Similarly, we computed the mean of the reduction time in each project when running version V3 and V1. The results are listed in the row starting with *V1-V3*.

According to the last two rows in Table III, the reduction in the mean time is very noticeable in GumTree. For example, for jEdit, the runtime of GumTree is reduced by 57.56 milliseconds by using the declaration-level pruning. The reduction rate is 81%. Furthermore, the runtime of GumTree is reduced by 60.09 milliseconds by using the statement-level pruning. The reduction rate is 85%.

For the four projects, on average, the mean running time of the original GumTree is reduced by at least 70% after using the declaration-level pruning, while it is reduced by at least 75% after using the statement-level pruning.

ChangeDistiller does not have as much runtime reduction as GumTree. Even in jDT Core, after using the declaration-level pruning, the total running time increased. Using the statement-level pruning, the total time of ChangeDistiller can reduce, by not as much as that in GumTree. We think it is because the average number of nodes in ChangeDistiller is much less than that in GumTree, which makes the time reduction is not much apparent.

C. The Correctness

To evaluate the correctness of the proposed pruning algorithm, we made comparisons between the changes returned by the original version and those returned by the pruned versions.

The results on ChangeDistiller are listed in Table IV. The numbers listed in the table are the number of files. In the table, the left part, i.e. column 2∼column 5, lists the results of comparing the original version with the declaration-level pruned version. In the right part, i.e. column 6∼9, lists the results of comparing the original version with the statement-level pruned version. The total number of files in each project is in Table I.

In Table IV, column *consistent* lists the number of files with the same changes returned by the two versions. The three sub-columns of column *inconsistent* list the number of files with different changes. The numbers listed in column *Total* is the total number of files with inconsistent changes. The numbers listed in column *Doc* are number of files in which the different changes are comments. Numbers listed in column *Other* are number of files in which the different changes are other kinds.

Take the first line for instance. According to Table I, the total number of files in jEdit is 13,830. In jEdit, after running the V1 version of ChangeDistiller and V2 version of ChangeDistiller on all of its files, 13,278 of them returned consistent results, accounting for 96%. For the remaining 552 files with inconsistent results, 288 of them have different changes that are comments related, i.e. removing or adding comments. The percentage is about 50%. After running the V1 version of ChangeDistiller and V3 version of ChangeDistiller on all of its files, 12,557 of them returned consistent results, accounting for 91%. For the remaining 1,273 files with inconsistent results, 564 of them have different changes that are comment related. It accounts for about 44%. Overall, after running the V1 version and V2 version of ChangeDistiller on all of the files in the dataset, 97% of them returned the same set of changes. For the

TABLE IV
COMPARING THE CHANGES RETURNED BY THE ORIGINAL CHANGEDISTILLER WITH THOSE RETURNED BY THE PRUNED VERSION. THE NUMBERS LISTED IN THE TABLE ARE THE NUMBER OF FILES.

Proj.	Comparing V1 with V2				Comparing V1 with V3			
	Consistent	Inconsistent			Consistent	Non Consistent		
		Total	Doc	Other		Total	Doc	Other
#1	13,278	552	288	264	12,557	1,273	564	709
#2	32,698	616	430	186	29,888	3,426	2,071	1,355
#3	10,477	292	163	129	9,957	812	384	428
#4	2,354	107	58	49	2,296	165	97	68
Total	58,807	1,567	939	628	54,698	5,676	3,116	2,560

remaining files, 60% of them have different changes that are comment related. After running the V1 version and V3 version of ChangeDistiller on all of the files in the dataset, 91% of them returned the same set of changes. For the remaining files, in 55% of them have different changes that are comment related.

1) Analyzing the Different Changes Between V1 and V2: Through browsing through the details of the different changes between the original version and the declaration-level pruned version, we found the following forms:

- When a constructor was inserted, removed, or modified, the original ChangeDistiller returns changes that are different from those returned by the pruned version. The following hunk illustrates such a case.

  ```
  private class BufferProps extends
  Hashtable{
  - BufferProps(Dictionary defaults){
  - this.defaults = defaults;
  - }
  ```

  In the example, an explicit constructor $BufferProps$ was removed. The original ChangeDistiller returns the following two changes:
  - PARAMETER_DELETE: defaults
  - STATEMENT_DELETE: this.defaults = defaults;

  The pruned version returns one change as shown below:
  - REMOVED_FUNCTIONALITY: BufferProps

  This is because the current pruning implementation only handles methods that occur explicitly in hunks. The default constructor $CInit()$ has not been processed yet. While in the original ChangeDistiller, when a constructor changes, it first matches the default constructor $CInit()$ with the changed constructor. In this example, ChangeDistiller parses the change as updating the default $CInit()$ with $BufferProps$. Therefore, it detected the two changes. However, the current pruning implementation only detected a removal of constructor $BufferProps$.
- Differences in processing fields. For example, the following three fields changed. One was removed and the other two were inserted.

  ```
  - private static Font font;
  + private static Vector recent;
  + private static int maxRecent;
  ```

  The original version interprets them as an insertion of $recent$ and an update of the $font$ to $maxRecent$, while the pruned version interprets them as an insertion of $maxRecent$ and an update of the $font$ to $recent$,

2) Analyzing the Different Changes Between V1 and V3: Through examining the details of the different changes between the original version and the statement-level pruned version, in addition to the two forms discussed above, we found another form as shown below:

- If a statement in a hunk is the same as the one in contextual lines, the changes returned by the original version may be different from those returned by the pruned version. For example, in the following code fragment, the statement $client.close()$ occurs in two places: one is in the hunk, and another is in the contextual try statement.

  ```
  if(auth != authInfo){
  + client.close();
  }
  try{
  client.close();
  }catch(IOException io){}
  ```

  The original ChangeDistiller detects a movement of the statement in the context to the hunk. Therefore, it returns the following changes:
  - STATEMENT_DELETE TRY_STATEMENT
  - STATEMENT_PARENT_CHANGE : client.close();
  - STATEMENT_INSERT TRY_STATEMENT
  - STATEMENT_INSERT: client.close();

  By contrast, the pruned version returns the following change:
  - STATEMENT_INSERT: client.close();

3) In the Case of GumTree: Table V lists the results of the consistency between changes returned by the original version of GumTree and those returned by the pruned versions.

According to the table, on average, after running the V1 version and the V2 version of GumTree on all of the files in the dataset, 75% of them returned the same set of changes. For the remaining files, 96% of them returned different changes that are comment related. After running the V1 version and the V3 version of GumTree on all of the files in the dataset, 70% of them returns the same set of changes. For the remaining files, 89% of them returns the different changes returned that are comment related.

TABLE V
COMPARING THE CHANGES RETURNED BY THE ORIGINAL GUMTREE WITH THOSE RETURNED BY THE PRUNED VERSION. THE NUMBERS LISTED IN THE TABLE ARE THE NUMBER OF FILES.

Proj.	Comparing V1 with V2				Comparing V1 with V3			
	Consistent	Inconsistent			Consistent	Non Consistent		
		Total	Doc	Other		Total	Doc	Other
#1	10,864	2,966	2,738	228	9,831	3,999	3,344	655
#2	27,227	6,087	5,762	325	25,583	7,731	6,722	1,009
#3	6,217	4,552	4,466	86	5,847	4,922	4,588	334
#4	1,193	1,268	1,245	23	1,109	1,352	1,287	65
Total	45,501	14,873	14,211	662	42,370	18,004	15,941	2,063

To sum up, both in ChangeDistiller and GumTree, the pruned version can return a high proportion of consistent changes with the original version. More than half of the inconsistent changes are comment related. Others are caused by the limitations in our current implementation.

D. Threats to Validity

We now discuss the threats to the validity of our evaluation.

Firstly, the runtime is susceptible to the underlying hardware environment and the implicit processes. Therefore, the runtime we calculated may not reflect the actual executing time of the

two tools, especially in the case of ChangeDistiller. Because the execution time of the original ChangeDistiller is already short. Any noise time may cause bias on the time we calculated. Therefore, the effectiveness of our comparisons between the runtime of the original version and that of the pruned versions is threaten by noise time, especially in the case of ChangeDistiller.

Secondly, when we compared the changes returned by the original GumTree and the pruned version, we compared the changes in the html format instead of the raw edit script. A raw GumTree edit script is a sequence of actions, which is usually long. Furthermore, when there are multiple changes in a file, their actions are entangled. It is not easy to identify which actions belong to which changes. Therefore, it is difficult to determine which changes output by the pruned version are consistent with those output by the original version. The html format GumTree changes are shorter and more abstract than the raw edit script. However, each html format change has an id for every mapping. When we ran the original and pruned versions, we found that many scripts differ only in the mapping ids and the locations of html tags. Therefore, we removed all mapping ids and html tags from the script. This may affect the validity of the consistency rate in the case of GumTree. In ChangeDistiller, each change is classified as a specific type. Before comparing the changes returned by the original version with those returned by the pruned version, we sorted the changes by line number and type. Therefore, the consistency rate calculation in ChangeDistiller is more reasonable.

In addition, we only considered the two Java based tree-differencing tools in the evaluation. This is a threat to the external validity. When applying the pruning to tree-differencing tools with other programming languages, the results may not be as good as in the evaluation.

VI. Related Work

Textual Differencing. The well-known GNU diff [1] can return added or removed lines, based on the longest common subsequence algorithm [12]. LDiff [2] and LHdiff [10] improve the GNU diff by detecting moved lines. However, textual-differencing approaches only return textual differencing lines or tokens, not structural changes.

Tree Differencing. Tree-differencing approaches return structural changes by comparing two ASTs representing two versions of a source file. The most famous algorithm is ChangeDistiller [3]. It detects changes in classes, methods, and fields. Furthermore, it classifies changes according to tree edit operations [13]. GumTree [4], Diff/TS [5], MTDIFF [11] and IJM [16] support moves, while CSLICER [14] and RTED [15] do not.

ChangeDistiller is highly efficient and our evaluation proves this. However, its granularity is limited to the statement-level. GumTree generates fine-grained edit scripts including moving actions. MTDIFF optimizes the detection of moved code parts. IJM generates more accurate and compact edit scripts than GumTree and MTDIFF. GumTree, MTDIFF and IJM use a strategy to identify identical subtrees and add pairs of identical subtrees directly to the mapping set. The strategy is based on comparing the fingerprints of two subtrees. Similarly, Diff/TS has a preprocessing step to prune the shared subtrees before generating the mappings. It is based on comparing the digest values of two subtrees. However, we trim ASTs by comparing the range of an entity with the range of hunks.

In JSync [17], the mapping consists of three phases: initial mapping, bottom-up mapping and top-down mapping. It first transforms two ASTs into two textual sequences of lines based on the textual value of the leaf nodes. Then, it gets initial mappings of the leaf nodes in the two ASTs using the longest common subsequence algorithm. The initial mapping is highly efficient due to the text-based approach. To some extent, this is like our approach. Because we use hunks to get the filter tree. Hunks are extracted by textual-differencing algorithms and are therefore efficient.

In CLDiff [18], unchanged declarations are pruned in a preprocessing to avoid unnecessary differencing analysis on unaltered AST nodes. However, statement nodes are not pruned.

VII. Conclusions

We propose to prune the matching tree to reduce the time of tree-differencing approaches. We have designed a pruning algorithm and integrated it to ChangeDistiller and GumTree. Through evaluation on four open projects, the declaration-pruned version of the two tools has at least a 64% reduction in the number of nodes, while the statement-pruned version of the two tools has at least a 74% reduction in the number of nodes.

The time on the original GumTree is reduced by at least 70% after using the declaration-level pruning, while is reduced by at least 75% after using the statement-level pruning. The time reduction in ChangeDistiller is not much as obvious as in GumTree. Because ChangeDistiller is limited to the statement-level changes, its average number of nodes in the matching tree is much less than that in GumTree.

The time can be further reduced if we generate the matching tree based on the filter tree directly. Because in the current implementation the AST is visited twice. In the first time, the AST is visited to generate the filter tree. In the second time, it is visited to generate the matching tree. In addition, the time on script generation did not decrease very much. Because in the declaration-level pruning, all of the statements in a method are still to be compared to generate mappings. In the statement-level pruning, even though not all of the statements involved, many contextual statements still keep in the tree.

Therefore, the next work includes two parts: Firstly, we plan to rewrite the tree-generation of ChangeDistiller and GumTree based on the filter tree. Secondly, we will study how to reduce the time on script generation.

Acknowledgment

The work was supported by the National Natural Science Foundation of China under Grant No.61502259.

REFERENCES

[1] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," Communications of the ACM, 1977, vol.20, no.5, pp.350-353.

[2] G. Canfora, L. Cerulo and M. Di Penta, "Ldiff: An enhanced line differencing tool," In Proceedings of the 31st International Conference on Software Engineering, 2009, pp. 595-598.

[3] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," IEEE Transactions on software engineering, 2007, vol.33, no.11, pp.725-743.

[4] J. R. Falleri, F. Morandat, X. Blanc, M. Martinez and M. Monperrus, "Fine-grained and accurate source code differencing," In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 313-324.

[5] M. Hashimoto and A. Mori, "Diff/TS: A Tool for Fine-Grained Structural Change Analysis," In WCRE'08: Working Conf. Reverse Eng., Antwerp, Belgium, 2008, pp. 279-288.

[6] E. Giger, M. Pinzger, and H. C. Gall, "Can we predict types of code changes? an empirical analysis," In 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, pp. 217-226.

[7] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," IEEE Transactions on Software Engineering, 2015, Vol. 31, No.6, pp.429-445.

[8] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change impact analysis based on a taxonomy of change types," In Proceedings of the 34th Annual Computer Software and Applications Conference (COMPSAC), 2010, pp. 373-382.

[9] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," In Proceedings of the 12th Working Conference on Mining Software Repositories, 2015, pp. 180-190.

[10] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta, " LHDiff: A language-independent hybrid approach for tracking source code lines," In 29th IEEE International Conference on Software Maintenance (ICSM 2013), 2013, pp. 230-239.

[11] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 660-671.

[12] E. W. Myers, "AnO (ND) difference algorithm and its variations," Algorithmica, 1986, Vol.1, No.1, pp.251-266.

[13] B. Fluri and H. C. Gall,"Classifying change types for qualifying change couplings," In Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC 2006, pp. 35-45.

[14] Y. Li, J. Rubin, and M. Chechik, "Semantic Slicing of Software Version Histories," In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, 2015, pp. 686-696.

[15] M. Pawlik and N. Augsten, "RTED: A Robust Algorithm for the Tree Edit Distance," In Proceedings of the VLDB Endowment, 2011, Vol. 5, No. 4, pp. 334-345.

[16] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating Accurate and Compact Edit Scripts Using Tree Differencing," In Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, 23-29 Sept. 2018, Madrid, Spain, pp. 264-274.

[17] H. A. Nguyen, T. T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen, "Clone Management for Evolving Software," IEEE Transactions on Software Engineering, 2012, vol.38, no.5, pp.1008-1026.

[18] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "CLDIFF: Generating Concise Linked Code Differences," In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, Montpellier, France, 2018, pp. 679-690