Reducing Mismatches in Syntax Coupled Hunks

Chunhua Yang[1,2], Xiufang Li[1]
[1]*School of Computer Science and Technology*
QILU University of Technology(Shandong Academy of Sciences)
[2] *Shandong Wiztek Science and Technology Co., Ltd.*
Jinan, China
jnych@126.com, lixf@qlu.edu.cn

***Abstract*—Hunks generated by textual-differencing tools are often used for understanding code changes. However, in the side-by-side view, the match between the deleted and added lines of a hunk is sometimes inconsistent with actual changes to the corresponding syntax entities. This mismatch usually occurs in syntax coupled hunks, i.e. hunks that contain changes to multiple syntax entities. It makes the hunks incomprehensible and misleading.**

A hybrid differencing algorithm is proposed to alleviate this problem. It applies tree-differencing to syntax coupled hunks to generate edits. It then maps edits back to the source code to generate adjusted hunks. Based on the current implementation, we conducted a case study on 10 open source projects. The results showed that 15% of commits contain syntax coupled hunks. And, we evaluated the results of the algorithm on 1,500 randomly drawn samples, and the correct matching rate was as high as 97%, demonstrating the effectiveness of the algorithm in reducing mismatches.

***Index Terms*—code differencing, hunks, mismatching, change understanding, software evolution**

I. Introduction

Understanding how software changes has become a regular part of modern software development. Many version management systems and IDEs provide differencing tools to present changes to the source code. The prevalent differencing tools are textual because they are efficient and not limited to programming languages. They return deltas by comparing the text values of the original and modified versions of the source code.

The common form of deltas is line-based hunks that are displayed in a unified view. For example, Fig.1 depicts two hunks returned by the well-known GnuDiff [1]. Each hunk consists of deleted lines(red), inserted lines(green), and surrounding contextual lines(white). Tools such as GitHub Diff [2], KDiff3 [3] and Mergely [4] also provide a side-by-side view, which can show the relationship between deleted lines and inserted lines more clearly. For example, Fig.2 is a split view provided by GitHub Diff to present the hunks in Fig.1.

However, the hunks generated by textual differencing tools are sometimes `syntax coupled`, that is, they contain changes to multiple syntax entities. For example, as shown in Fig.1, changes to methods *raiseTimeoutFailure* and *performOnPrimary* scattered and entangled in two hunks. This is a two-to-one method coupling, where two methods in the original version are coupled with one method in the modified version.

In syntax coupled hunks, mismatch is a common phenomenon, usually in the following two forms:

1) Mismatches between nodes in the original and modified versions of the source code. For instance, according to the hunks shown in Fig.2, the *raiseTimeoutFailure* method in the original version matches *performOnPrimary* method in the modified version. But obviously, the *performOnPrimary* method of the same name in the original and modified versions should match.
2) Mismatches between non-code lines (such as delimiters and comments), or certain elements of a signature or statement (such as arguments and annotations). For example, in Fig.2, the brace on line 543 of the original version matches the brace on line 509 of the modified version incorrectly. In fact, the two curly braces are a necessary part of the deleted If-statement and the inserted If-statement respectively, so they should not be recognized as context.

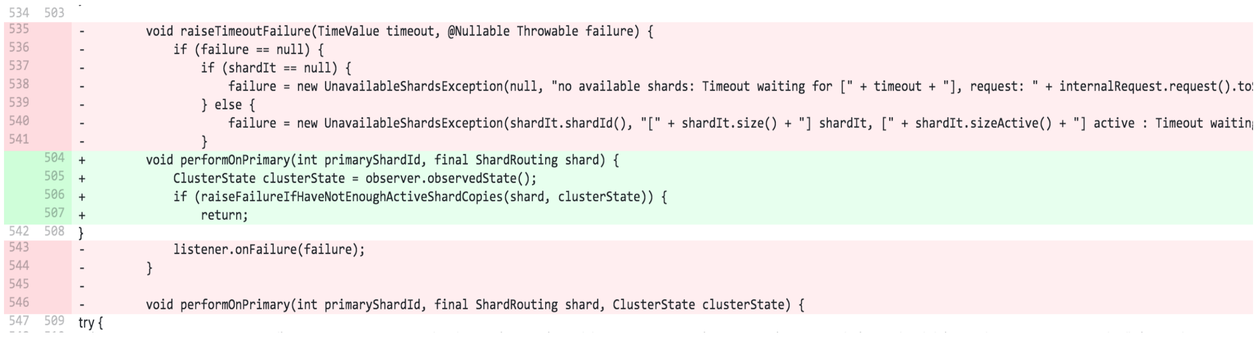
Mismatches in syntax coupled hunks hinders change understanding in the following ways:

- Mismatches between nodes can mislead users with incorrect edit operations, thereby obscuring the actual changes. For example, according to Fig.1, the edit operation between *raiseTimeoutFailure* and *performOnPrimary* is an update. But, the update should be actually between *performOnPrimary(int, ShardRouting, ClusterState)* of the original version and *performOnPrimary(int, ShardRouting)* of the modified version.
- Mismatches cause changes to the entire entity to be spread out over multiple hunks, making these hunks incomprehensible. In order to find out the actual changes, the user must go through all deleted and inserted lines.

In order to alleviate the mismatch problem in syntax coupled hunks, an algorithm is porposed in the paper. It makes the following contributions.

- It is a novel hybrid differencing algorithm for alleviating the mismatch problem. It applies tree-differencing to syntax coupled hunks to generate edit operations. It then maps edit operations back to the source code to generate adjusted hunks.

DOI reference number: 10.18293/SEKE2022-049



```diff
-         void raiseTimeoutFailure(TimeValue timeout, @Nullable Throwable failure) {
-             if (failure == null) {
-                 if (shardIt == null) {
-                     failure = new UnavailableShardsException(null, "no available shards: Timeout waiting for [" + timeout + "], request: " + internalRequest.request().to
-                 } else {
-                     failure = new UnavailableShardsException(shardIt.shardId(), "[" + shardIt.size() + "] shardIt, [" + shardIt.sizeActive() + "] active : Timeout waitin
-                 }
+         void performOnPrimary(int primaryShardId, final ShardRouting shard) {
+             ClusterState clusterState = observer.observedState();
+             if (raiseFailureIfHaveNotEnoughActiveShardCopies(shard, clusterState)) {
+                 return;
 }
-             listener.onFailure(failure);
-         }
-
-         void performOnPrimary(int primaryShardId, final ShardRouting shard, ClusterState clusterState) {
 try {
```

Fig. 1. An Example of Syntax Coupled Hunks.

- In addition to hunks, the algorithm outputs edit operations that facilitate analysis. For example, the hunk-level change dependency analysis becomes feasible. So far, change dependency analysis has been mainly performed at the method or line level [5] [6].

We have implemented the algorithm and conducted a case study to examine the distribution of syntax coupled hunks and evaluate the effectiveness of the algorithm.

The remainder of the paper is organized as follows. In Section 2, we present the algorithm. In Section 3, we present the implementation and case study. We review the related work in Section 4 and summarize the paper in Section 5.

II. The Algorithm

Syntactically, changes to the source code are edit operations on the nodes of the abstract syntax tree (AST). Typical edit operations include insert, delete, update, and move. If one or more hunks contain multiple edit operations on nodes on the same level of the AST, they are syntax coupled. For example, the two syntax coupled hunks in Fig.1 contain a deletion of method *raiseTimeoutFailure* and an update on method *performOnPrimary*.

We define a tuple (H, N_1, N_2) to represent syntax coupled hunks, where H is a set of hunks, N_1 and N_2 are the sets of nodes in the original and modified versions of AST whose changes occur in the hunks of H.

Inputting syntax coupled hunks, the algorithm generates the adjusted hunks through a differencing phase and a layout phase. During the differencing phase, tree-differencing is applied to the nodes in both versions to generate edit operations. Then, in the layout phase, the lines of code belonging to these edit operations are sorted, and the remaining lines are filled in the appropriate positions to produce adjusted hunks.

A. The Tree-Differencing Phase

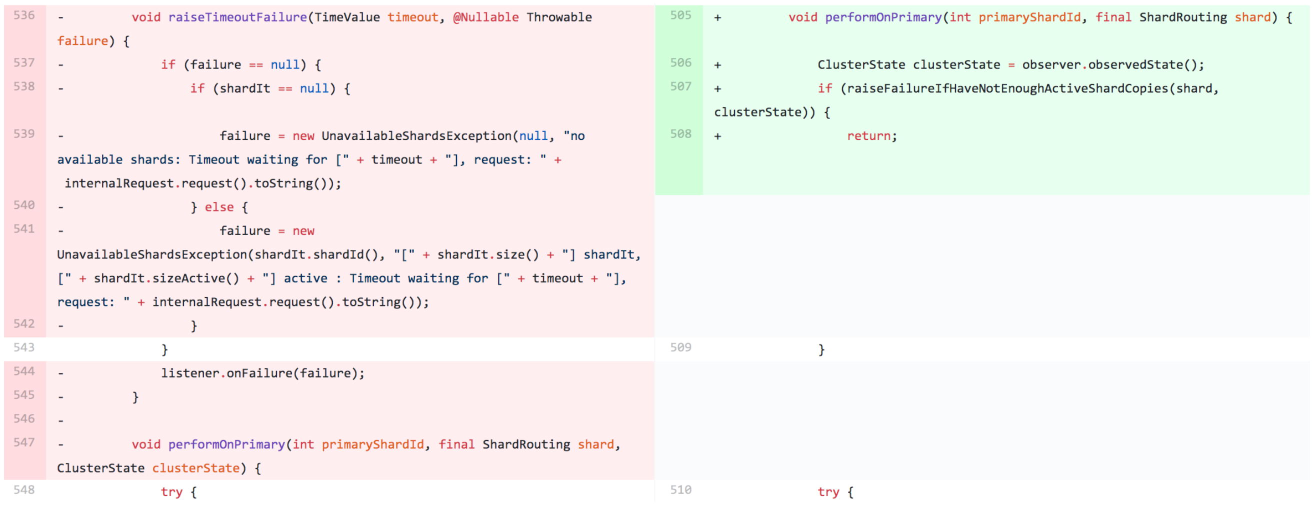
The process is described in Algorithm 1. The algorithm inputs a hunk set H and two node sets N_1 and N_2 that belong to the original and modified versions of AST, and outputs edit operations. The main steps are as follows:

- Firstly, through the *Matching* function, the nodes in N_1 and N_2 are compared with each other to find similar nodes. (Line 1)

Algorithm 1: HASTDiff(H, N_1, N_2)

```
Input: a set $H$ of hunks, two sets of nodes $N_1$ and $N_2$
Output: The set of edit operations $O$
1  $P \leftarrow Matching(N_1, N_2)$;
2  $Chr_1 \leftarrow \varnothing$; $Chr_2 \leftarrow \varnothing$; $O \leftarrow \varnothing$;
3  for each pair $(n_1, n_2) \in P$ do
4      $O \leftarrow O \cup genOp(n_1, n_2)$;
5      $C_1 \leftarrow childrenInHunks(n_1, H)$;
6      $C_2 \leftarrow childrenInHunks(n_2, H)$;
7      $H_c \leftarrow hunksCrossingNodes(C_1) \cup$
         $hunksCrossingNodes(C_2)$;
8      $U \leftarrow hunkgroupExtract(H_c, C_1, C_2)$;
9      for each pair $(H_U, N_{U1}, N_{U2}) \in U$ do
10         $O \leftarrow O \cup HASTDiff(H_U, N_{U1}, N_{U2})$;
11     $Chr_1 \leftarrow Chr_1 \cup unmatchedNodes(C_1, U)$;
12     $Chr_2 \leftarrow Chr_2 \cup unmatchedNodes(C_2, U)$;
13 $N_{unmatched1} \leftarrow unmatchedNodes(N_1, P)$;
14 for each node $n_1 \in N_{unmatched1}$ do
15     $O \leftarrow O \cup genOp(n_1, null)$;
16     $Chr_1 \leftarrow Chr_1 \cup childrenInHunks(n_1, H)$;
17 $N_{unmatched2} \leftarrow unmatchedNodes(N_2, P)$;
18 for each node $n_2 \in N_{unmatched2}$ do
19     $O \leftarrow O \cup genOp(null, n_2)$;
20     $Chr_2 \leftarrow Chr_2 \cup childrenInHunks(n_2, H)$;
21 $U \leftarrow hunkgroupExtract(H, Chr_1, Chr_2)$;
22 for each pair $(H_U, N_{U1}, N_{U2}) \in U$ do
23     $O \leftarrow O \cup HASTDiff(H_U, N_{U1}, N_{U2})$;
24 for each node $n_1 \in unmatchedNodes(Chr_1, U)$ do
25     $O \leftarrow O \cup genOp(n_1, null)$;
26 for each node $n_2 \in unmatchedNodes(Chr_2, U)$ do
27     $O \leftarrow O \cup genOp(null, n_2)$;
```

- Then, for each matched node pair, an *update* operation is generated. And, their children in common hunks are recursively differentiated. Their remaining children are added to the sets Chr_1 and Chr_2, respectively. (Line 3∼12)
- For each unmatched node in the set N_1 or N_2, a *delete* or



```
-        void raiseTimeoutFailure(TimeValue timeout, @Nullable Throwable failure) {
-            if (failure == null) {
-                if (shardIt == null) {
-                    failure = new UnavailableShardsException(null, "no available shards: Timeout waiting for [" + timeout + "], request: " + internalRequest.request().toString());
-                } else {
-                    failure = new UnavailableShardsException(shardIt.shardId(), "[" + shardIt.size() + "] shardIt, [" + shardIt.sizeActive() + "] active : Timeout waiting for [" + timeout + "], request: " + internalRequest.request().toString());
-                }
             }
-            listener.onFailure(failure);
-        }
-
-        void performOnPrimary(int primaryShardId, final ShardRouting shard, ClusterState clusterState) {
             try {
```

```
+        void performOnPrimary(int primaryShardId, final ShardRouting shard) {
+            ClusterState clusterState = observer.observedState();
+            if (raiseFailureIfHaveNotEnoughActiveShardCopies(shard, clusterState)) {
+                return;
             }
             try {
```

Fig. 2. The Hunks Generated by GitHub Diff for the Hunks in Fig.1.

```
Algorithm 2: Layout($O$, $H$)
  Input: a set $O$ of edit operations, and a set $H$ of hunks
  Output: The set of adjusted hunks $L$
1  $O_{sorted} \leftarrow \varnothing$;
2  $L_1 \leftarrow sortOps(O, true)$;
3  $L_2 \leftarrow sortOps(O, false)$;
4  $insertContextRanges(L_1, H)$;
5  $insertContextRanges(L_2, H)$;
6  while $hasNext(L_1) \wedge hasNext(L_2)$ do
7      $(S_1, u_1) \leftarrow findNextUpdateOrContext(L_1)$;
8      $(S_2, u_2) \leftarrow findNextUpdateOrContext(L_2)$;
9      $O_{sorted} \leftarrow O_{sorted} \cup S_1$;
10     $O_{sorted} \leftarrow O_{sorted} \cup S_2$;
11     $addUpdateOrContext(O_{sorted}, u_1, u_2)$;
12 $addIsolatedLines(O_{sorted}, H)$;
13 $matchingIsolatedBraces(O_{sorted})$;
14 $L \leftarrow genAdjustedHunks(O_{sorted}, H)$
```

insert operation will be generated, and then its children will be added to Chr_1 and Chr_2 respectively. (Line 13∼20)

- A recursive differencing is applied to the children in sets Chr_1 and Chr_2. Then, a *delete* or an *insert* operation is generated for each unmatched child in Chr_1 or Chr_2, respectively. (Line 21∼27)

For syntax coupled nodes, the function *Matching* is used to determine which node in the original version is most similar to which node in the modified version. We use a strategy similar to that used in ChangeDistiller [12] to check the similarity between nodes. The function *hunkgroupExtract* is used in the algorithm to extract the syntax coupled hunk groups. And, after all edit operations are generated, we generate an *ADD* operation for consecutive *insert* operations that belong to an entire entity, and a *DEL* operation for consecutive *delete* operations that belong to an entire entity.

B. The lay out phase

In this phase, the edit operations generated in the previous phase will be laid out to generate adjusted hunks. The resulting hunks are presented in a split view.

All operations are sorted in ascending order by the starting line number of the operation. Lines of code in the original and modified versions belonging to the *update* operation are displayed horizontally. And the update operations and context lines are set as boundaries to arrange other operations and the remaining hunk lines.

Algorithm 2 depicts the process. The main steps are as follows:

1) The operations in the set O are first sorted by the line number of the nodes in the original version(line 2) and the modified version(line 3), respectively. The sorted operations are stored in lists L_1 and L_2, respectively.
2) Then the context lines between the hunks in H are inserted into appropriate places in the lists L_1 and L_2, respectively. (Line 4∼5)
3) Next, for the lists L_1 and L_2, repeat the following steps until both lists are empty:
   a) Find the next update operation or context from each list. (Line 7∼8). The function *findNextUpdateOrContext* implements it. For a list L, the function returns the update operation or context u, and a list S of operations before u.
   b) Insert the operations of S_1 into the list O_{sorted}. Then insert the operations of S_2. (Line 9∼10).
   c) Finally, insert u_1 and u_2 into O_{sorted}, respectively. (Line 11)
4) Insert the remaining deleted or inserted lines properly into O_{sorted}. (Line 12)
   In this step, if there are isolated braces in the original and modified versions, they will be matched properly. The matching braces become the context.

5) Finally, the adjusted hunks are generated according to the lines arranged in the O_{sorted} list.

C. Illustration

Take Fig.1 as an example. Through the tree-differencing, the signature *performOnPrimary(int, ShardRouting, ClusterState)* in the original version matches the signature *performOnPrimary(int, ShardRouting)* in the modified version. And, the two children of the method *performOnPrimary* in the modified version, namely the local declaration and the If-statement, are identified as insert. Since the If-statement and its children are identified as insert operations, the changes to the If-statement are identified as an ADD.

Unmatched signature *raiseTimeoutFailure(TimeValue, Throwable)* and its children are identified as delete operations. Since its signature and its children are identified as delete operations, the method is identified as DEL.

Therefore, through the differencing phase, the following operations will be generated.

- *DEL raiseTimeoutFailure*(Line 535∼544)
- *update performOnPrimary* (Line 546 of the original version and Line 504 of the modified version)
- *ADD Local declaration* (Line 505)
- *ADD if statement* (Line 506∼508)

In the layout phase, these four operations are sorted. And the remaining isolated lines (the deleted line 545) are inserted before the update operation. The output of the algorithm is shown in Fig.3.

III. The Implementation and Case Study

We have implemented the algorithm. We use our previous work [7] to extract the syntax coupled hunk groups and use the proposed algorithm to generate the adjusted hunks in each group.

Based on the current implementation, we conducted a case study. The aim of the case study is to examine the distribution of syntax coupled hunks and evaluate the effectiveness of the algorithm.

To achieve these aims, the following research questions are to be answered:

- **RQ1:** What is the proportion of syntax coupled hunks in daily revisions?
- **RQ2:** Does the proposed algorithm generate the correctly-matched results for syntax coupled hunks?

A. The Data Set

We selected 10 open java projects from GitHub. They have different periods, stars, and scales.

The information of these projects is listed in Table I. For example, since 2010, the most popular project *elasticsearch* has 36,918 commits. Note that, we only list the number of commits containing code change hunks. Commits with only non-code changes were ignored.

TABLE I
Study Projects and Their Total Number of Revisions($Commits$) That Contain Code Change Hunks.

No.	Project	Stars	Peroid	Commits
1	activemq	1.6k	2005-12-12∼2019-11-5	5,916
2	eclipse.jdt.core	166	2001-6-5 ∼2019-11-6	15,150
3	elasticsearch	45.3k	2010-2-8 ∼2019-10-4	36,918
4	glide	27.6k	2012-12-20 ∼2019-11-6	1,625
5	guice	8.7k	2006-8-22 ∼2019-10-4	877
6	hibernate-orm	4.1k	2004-6-3 ∼2019-10-8	7,413
7	jEdit	17	2001-9-2 ∼2019-10-15	4,998
8	maven	1.9k	2003-9-1 ∼2019-11-5	5,388
9	redisson	11.1k	2013-12-22 ∼2019-11-6	2,461
10	spring-framework	33.4k	2008-7-10 ∼2019-11-5	12,930
Total				93,676

B. The Distribution of Syntax Coupled Hunks(RQ1)

For convenience, we use *hunk group*s to represent the syntax coupled hunk groups. Using the current implementation, we extracted hunk groups in each project and calculated the number of hunk groups at each granularity.

Table II lists the number of commits that contain hunk groups, the total number of hunk groups, and the number of hunk groups at each granularity for each project. We can see that 14,357 commits in the dataset contain hunk groups, which account for 15% of the total commits(as shown in Table I). In addition, the number of hunk groups with method level coupling ranks first in each project. The number of hunk groups with statement level coupling ranks second, except for Guice and Hibernate-orm. In Guice and Hibernate-orm, the class level coupling ranks second.

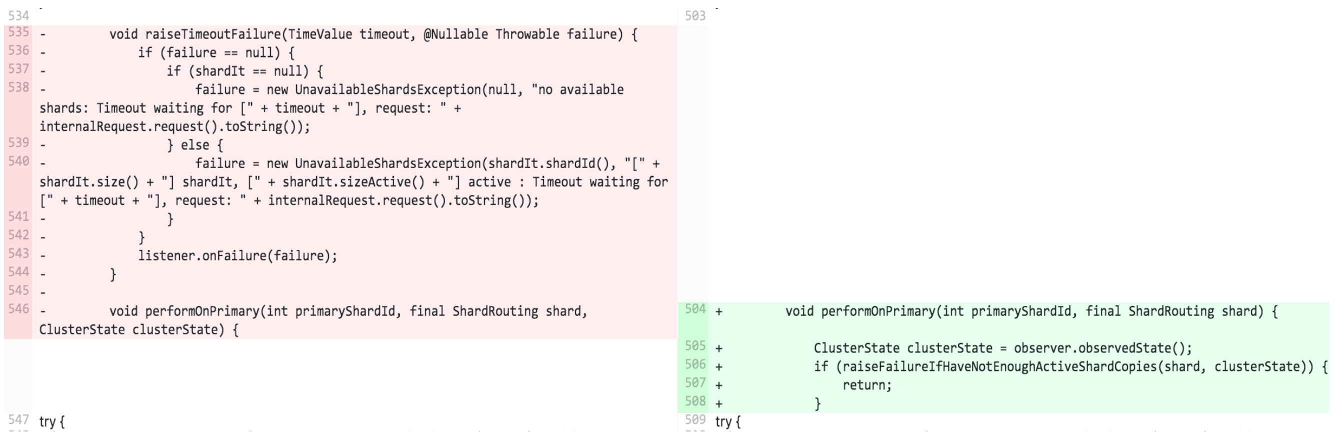
TABLE II
The Number of Commits With Hunk Groups Per Project, and the Number of Hunk Groups at Each Granularity. H_{class}, H_{method}, H_{field}, and H_{stmt} Represent the Number of Hunk Groups at Class, Method, Field, and Statement Granularity, Respectively.

Prj.	Commits	Number of Syntax Coupled Hunk Groups				
		Total	$\mathbf{H}_{class}$	$\mathbf{H}_{mthod}$	$\mathbf{H}_{field}$	$\mathbf{H}_{stmt}$
1	653(11%)	955	16	608	24	323
2	2,055(14%)	4,892	43	3,349	275	1,336
3	5,731(16%)	11,155	744	8,703	355	1,788
4	269(17%)	507	88	433	25	48
5	143(16%)	221	45	175	10	26
6	1,327(18%)	3,211	131	2,778	83	369
7	758(15%)	1,314	62	844	51	410
8	806(15%)	1,227	19	825	98	322
9	375(15%)	730	3	663	5	64
10	2,240(17%)	4,708	289	3,747	55	711
Total	14,357(15%)	28,920	1,440	22,125 (77%)	981	5,397 (19%)

RQ1: In summary, 15% of code change commits contain syntax coupled hunks.

C. Effectiveness of The Proposed Algorithm(RQ2)

In order to answer the second research question, based on a set of samples randomly selected from the dataset, we



```
-        void raiseTimeoutFailure(TimeValue timeout, @Nullable Throwable failure) {
-            if (failure == null) {
-                if (shardIt == null) {
-                    failure = new UnavailableShardsException(null, "no available
shards: Timeout waiting for [" + timeout + "], request: " +
internalRequest.request().toString());
-                } else {
-                    failure = new UnavailableShardsException(shardIt.shardId(), "[" +
shardIt.size() + "] shardIt, [" + shardIt.sizeActive() + "] active : Timeout waiting for
[" + timeout + "], request: " + internalRequest.request().toString());
-                }
-            }
-            listener.onFailure(failure);
-        }
-
-        void performOnPrimary(int primaryShardId, final ShardRouting shard,
ClusterState clusterState) {
try {
```

```
+        void performOnPrimary(int primaryShardId, final ShardRouting shard) {

+            ClusterState clusterState = observer.observedState();
+            if (raiseFailureIfHaveNotEnoughActiveShardCopies(shard, clusterState)) {
+                return;
+            }
try {
```

Fig. 3. The Adjusted Hunk Generated by the Proposed Algorithm for the Hunks in Fig.1.

manually evaluated the results generated by the proposed algorithm. Meanwhile, we compared the results with those generated by GitHub Diff and Mergely. We adopt these two tools due to the following reasons:

- GitHub is a platform that contains lots of open source projects. As a result, its diff is widely used for change understanding.
- Mergely provides a special side-by-side diff view style. And it provides a JS library. So, based on a hunk group set, it is easy to run Mergely on it and display the results in an html page.

For each sample hunk group, we checked the hunks generated by the algorithm, GitHub Diff and Mergely, to see if they matched correctly. We created a web page to view samples and evaluate results. The authors and three master students did the manual assessment. They have rich experience in software development. The three students first evaluated the samples of different projects independently. Then, the author reviewed their work.

We considered the coupling structure and the number of hunks contained when selecting samples. In the end, a total of 1,437 samples were selected. The left half of Table III lists the numbers of samples selected from each project. The evaluation results of the sample set are shown in the right half of table III. According to the table, the algorithm correctly matched 1,395 hunk groups, accounting for 97%. In the case of GitHub Diff and Mergely, the figures are 1,213 and 1,106, accounting for 84% and 77%, respectively.

RQ2:To sum up, in 97% of the samples, the adjusted hunks generated by the proposed algorithm are correctly matched.

The correct matching rate is higher than GitHub Diff and Mergely.

D. Threats to Validity

As the current implementation is based on Java, all selected projects are written in Java. In addition, since manual evaluation is time-consuming, the number of samples selected is not large. As a result, the validity of the case study is threatened by the programming language and sample size of the selected

TABLE III
THE NUMBER OF SAMPLE HUNK GROUPS SELECTED AND THE EVALUATION RESULTS. HAST REPRESENTS THE PROPOSED ALGORITHM, GH REPRESENTS GITHUBDIFF, AND MG REPRESENTS MERGELY.

Prj.	**Number of Sample Hunk Groups**				**Correct Matching**		
	H_{class}	H_{method}	H_{stmt}	**Total**	**HAST**	**GH**	**MG**
1	2	119	48	169	161	149	136
2	5	65	64	134	131	123	120
3	69	169	67	305	295	263	239
4	14	73	8	95	91	77	69
5	20	33	4	57	54	51	53
6	22	89	34	145	142	117	107
7	11	51	35	97	96	75	74
8	7	94	38	139	135	98	90
9	2	102	16	120	118	107	82
10	33	104	39	176	172	153	136
	185	899	353	1,437	1,395	1,213	1,106
					97%	84%	77%

projects. In addition, the results of manual evaluation are influenced by inspectors. In some cases, whether one entity should be considered a match with another entity may vary from person to person.

IV. RELATED WORK

Textual Differencing. Textual-differencing tools detect text changes based on the longest common subsequence algorithm [8]. They usually generate line-based hunks(i.e., deltas). The well-known GNU diff can return added or deleted lines, which has been widely integrated into IDEs and version-control systems to calculate and present source code changes. Tools such as LDiff [9] and LHdiff [10] improve the GNU diff by detecting moved lines. To make diff easier to read, tools like GitHub Diff, Mergely and KDiff3 provide a side-by-side view and the within-line differencing to refine changes in the hunk. In [11], it is shown that Git diff with different algorithm options can give different results and revealed that the Histogram option is better for describing code changes that the default Myers option. However, for the example shown in

Fig.2, the results of the two algorithm options are the same. The mismatch problem is not improved.

Tree Differencing. Tree-differencing approaches return structural changes by comparing two ASTs representing two versions of the source code. They generate edit operations that represent changes to syntax entities.

The most famous tree-differencing algorithm is ChangeDistiller [12]. It detects changes in classes, methods, and fields.

Diff/TS [13] detected changes in various syntax granularities including classes, statements and expressions. It can detect the move actions. GumTree [14] improved ChangeDistiller by producing a shorter edit script. JSync [15] and srcDiff [16] used the longest common subsequence algorithm to compare AST nodes. MTDIFF [17] improved the move actions and generated shorter edit scripts than Gumtree, RTED, JSync, and ChangeDistiller with a higher accuracy. Higo et al. [18] considered copy-and-paste as an editing action. IJM [19] can generate more accurate move and update operations than GumTree and ChangeDistiller. CLDiff [20] aimed at generating concise linked differences. Tree-differencing approaches generate syntax edits. However, they are less efficient than textual-differencing. In [21], a hybrid method was proposed to improve GumTree matching to generate shorter edit scripts. The matching algorithm in GumTree was enhanced by using line-based textual differencing. The method is a hybrid, similar to ours. However, they focused on the optimization of GumTree, not the mismatching problem in hunks.

Tangled changes. Tao and Kim [5] proposed to partition composite code changes by grouping static related changes and methods with similar names. Herzig and Zeller [6] proposed CONFVOTERS that combine various dependencies to detect the related changes. Barnett et al. [22] proposed CLUSTERCHANGES that can relate separate regions of change by using static analysis. However, these researches focused on the tangled changes that accomplish the same task. The algorithm proposed in the paper focuses on the tangled entities in hunks.

V. Conclusions

We present an algorithm that applies tree-differencing to syntax coupled hunks to alleviate the mismatch problem. Based on current implementation, we conducted a case study to examine the distribution of syntax coupled hunks and evaluate the effectiveness of the algorithm. We found that 15% of commits contained syntax coupled hunks. And, the proposed algorithm greatly improves the mismatching in syntax coupled hunks.

Since the algorithm is based on the textual-differencing and tree-differencing is only used to compare the nodes that cross hunks, it is efficient than tree-differencing. Therefore, it can be used to extend the textual-differencing tools to reduce mismatches without worrying about efficiency. As a future work, we will provide implementations in other programming languages.

Acknowledgment

This work is supported by Shandong Provincial Natural Science Foundation (Grant No. ZR2020MF031).

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] "GitHub Diff," https://github.com

[3] "KDiff3," http://kdiff3.sourceforge.net/

[4] "Mergely," http://www.mergely.com/

[5] 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.

[6] K. Herzig and A. Zeller. "The impact of tangled code changes," In Proceedings of the10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, 2013, pp.121-130.

[7] C. Yang, J. Whitehead, "Pruning the AST with Hunks to Speed up Tree Differencing," In Proceedings of 26th IEEE International Conference on Software Analysis, Evolution and Reengeneering (SANER), 2019, Hangzhou, China, pp. 15-25.

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

[9] 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.

[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] Y.S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in Git?," Empirical Software Engineering,2020, 25, pp.790-823.

[12] 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.

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

[14] 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.

[15] H. A. Nguyen, T. T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen, "Clone Management for Evolving Software," IEEE Trans. Softw. Eng., 38(5):1008-1026, Sep. 2012.

[16] M.J. Decker, M.L. Collard, L.G. Volkert, J.I. Maletic. "srcDiff: A syntactic differencing approach to improve the understandability of deltas," Journal of Software: Evolution and Process. 2020, 32(4).

[17] 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.

[18] Y. Higo, A. Ohtani, and S. Kusumoto, "Generating simpler AST edit scripts by considering copy-and-paste," In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 532-542.

[19] 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.

[20] 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

[21] J. Matsumoto, Y. Higo and S. Kusumoto, "Beyond GumTree: A Hybrid Approach to Generate Edit Scripts," In Proceedings of IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 550-554.

[22] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," In Proceedings of the 37th International Conference on Software Engineering, 2015, Vol. 1, pp.134-144.