2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

DiffFix: Incrementally Fixing AST Diffs via Context and Type Information

Guofeng Zeng[†], Chang-ai Sun[†*], Kai Gao[†], Huai Liu[‡]
[†] School of Computer and Communication Engineering, University of Science and Technology Beijing, China
[‡] Department of Computing Technologies, Swinburne University of Technology, Australia
d202110394@xs.ustb.edu.cn, casun@ustb.edu.cn, kai.gao@ustb.edu.cn, hliu@swin.edu.au

***Abstract*—The abstract syntax tree differencing (ASTDiff) technique aims to capture syntactic code changes through comparing the differences between a pair of ASTs of a program, which has been widely used in various program analysis or testing tasks, such as code review, clone detection, and regression testing. A key issue for ASTDiff lies in the accurate mappings between nodes of two ASTs. However, most existing approaches often fail to generate such perfect diffs due to the gap between diverse code changes and unsound node matching heuristics. Our in-depth investigation reveals that most inaccurate mappings are caused by the ignorance of context- and/or type-specific constraints. Accordingly, we propose an AST diff fixing approach DiffFix that leverages both the node's context and type constraints to iteratively and incrementally fix imperfect diffs. Comprehensive experiments have been conducted to evaluate the effectiveness of DiffFix through its application to fix diffs generated by five state-of-the-art ASTDiff techniques. The experimental results demonstrate that DiffFix can improve the perfect diff rate of these baseline techniques by 5.25% to 51.12% with negligible time overhead.**

***Index Terms*—source code differencing, abstract syntax tree, quality assurance, software evolution**

I. Introduction

Understanding code changes is essential for many software development and maintenance tasks, such as code review, debugging, and refactoring [1]. To this end, abstract syntax tree differencing (briefly ASTDiff) techniques [2]–[7] exhibit their superiority in capturing syntactic and structural changes by matching nodes and computing differences (i.e., edit actions) between ASTs of two versions of source code. In recent years, they have been successfully applied in numerous tasks, such as change pattern mining [8], [9], automated program repair [10], [11], code review assistance [12], and API usage adaptation [13]. Therefore, it is critical for ASTDiff techniques to generate diffs or node mappings that accurately reflect the developer's code change intentions.

Existing approaches typically use heuristics to match nodes for scalability, leading to the incapability of handling subtle cases of diverse real-world code changes. For example, a common heuristic is to match nodes using the mapping status of related nodes [3], [5]. However, such information is unsound (i.e., potentially inaccurate) and transient during node matching, which can result in inaccurate mappings. Other information such as AST node types and code change practices (e.g., refactoring) also hinders the node matching. These constraints are often ignored by general approaches [2], [3], [5] and are only partially utilized in language-specific approaches [7], [14], which cannot prevent inaccuracies either. Furthermore, the trade-off between accuracy and efficiency is also difficult to handle, and some improvements on node matching incur high overhead [6].

Prior work has shown that the accuracy of node mappings generated by existing techniques is far from satisfactory. Fan et al. [15] evaluated the accuracy of statement mappings and token mappings generated by three ASTDiff techniques, including (GumTree [3], MTDiff [5], and IJM [14]), and found that they generate inaccurate mappings for 20%-30% of the file versions. Alikhanifard et al. [7] created the first node mapping benchmark *DiffBenchmark* that consists of human-verified node mappings for 987 real-world commits. They evaluated the capability of four well-known techniques, including GumTree [3], MTDiff [5], IJM [14] and GumTree-simple [4], in generating perfect diffs (meaning that the node mappings generated by the technique are identical to the benchmark). They found that the rates of perfect diffs generated by all four techniques were below 64%.

After the investigations on inaccurate mappings generated by existing approaches, we summarize the three observations: i) *context-related*: most of the inaccurate mappings are related to the context of accurately mapped nodes, of which four types are dominant, including parent context, children context, inner context (relations of nodes within the same block), and nearby context (relations of adjacent statements); ii) *type-related*: most of the inaccurate mappings are of certain types, which can be classified into tokens, expressions, and statements, reflecting the constraints and difficulties to match nodes with specific syntax types; iii) *coexisting*: some diffs involve complex or numerous code changes, where different types of inaccurate types coexist or even interact with each other.

Based on the above findings, we propose a post-processing approach *diff fixing* that identifies and fixes inaccurate node mappings generated by ASTDiff techniques. In particular, we propose DiffFix, an incremental context and type-guided AST diff fixing approach, which consists of a warm-up phase and a fixing phase. The warm-up phase performs several initialization operations to speed up the fixing phase, including pre-fetching global and local information as well as maintain-

[*] Corresponding author: Chang-ai Sun

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00237

ing pruned node sets needed to analyze in the fixing phase. The fixing phase refines node mappings incrementally and iteratively across nine passes. Each pass involves traversing a pruned node set, identifying potential inaccurate mappings using specific context information, and fixing those inaccuracies with node type information. In the approach, we consider a diverse range of node types, such as variables, method invocations, infix expressions, and various statements.

To assess the effectiveness and efficiency of DIFFFIX, we have conducted a series of experiments on DiffBenchmark, an AST node mapping benchmark. We apply DIFFFIX to five state-of-the-art ASTDiff techniques, including RM-ASTDiff [7], iASTMapper [6], DiffAutoTuning [16], GumTree-simple [4], and GumTree [3], and compare their performances before and after applying DIFFFIX. The experimental results show that DIFFFIX can effectively fix inaccurate mappings, significantly increase the rate of perfect diffs, and reduce edit script size on these techniques (to which different fix passes contribute); meanwhile, the overhead of DIFFFIX is negligible compared to the node matching phase, thanks to the 15x$\sim$20x speedups brought by optimization strategies.

This study makes the following contributions:

- We propose an incremental context and type-guided AST diff fixing approach DIFFFIX that can be generally applied to various ASTDiff techniques. DIFFFIX leverages diverse context and type information to identify and fix inaccurate mappings in diffs generated by these techniques.
- We conduct a comprehensive evaluation on DiffBenchmark with five state-of-the-art ASTDiff techniques in terms of accuracy, efficiency, and edit script size, demonstrating the high performance of DIFFFIX.
- We publicize our code and data at https://github.com/guofeng99/DiffFix to facilitate future research.

The rest of this paper is structured as follows: Section II introduces the background and related work. Section III discusses motivating examples. Section IV presents our approach DIFFFIX. Section V reports an evaluation of DIFFFIX. Section VI discusses the main implications of this work and impacts on downstream tasks. Finally, Section VII concludes the paper with providing the future work.

II. BACKGROUND AND RELATED WORK

A. AST Differencing

AST is a tree-like program representation generated by language parsers to support program compilation and lightweight program analysis [17]–[19]. Each AST node has a type field to indicate the syntactic structure and an optional value field to record its corresponding token.

A typical ASTDiff process includes three phases:

a) AST Parsing: This phase parses two versions of a program into AST_1 and AST_2. As shown in Fig. 1, there is one changed line (marked in red and green) between the two file versions on the left side. Their corresponding ASTs are on the middle side (only the parts of the changed line are shown).

b) Node Matching: This phase maps nodes between AST_1 and AST_2 (i.e., an AST diff) based on heuristics, following two principles from traditional tree differencing [20]: nodes in AST_1 and AST_2 are either mapped in only one mapping or unmapped; the nodes in a mapping have the same type. As shown in the middle of Fig. 1, the expected mapped nodes are marked with the same number. The middle on the right side of Fig. 1 shows the mappings of two changed nodes. Node matching is the most challenging part of ASTDiff, both in terms of being accurate enough to reflect code change intentions and efficient enough to analyze large-scale code.

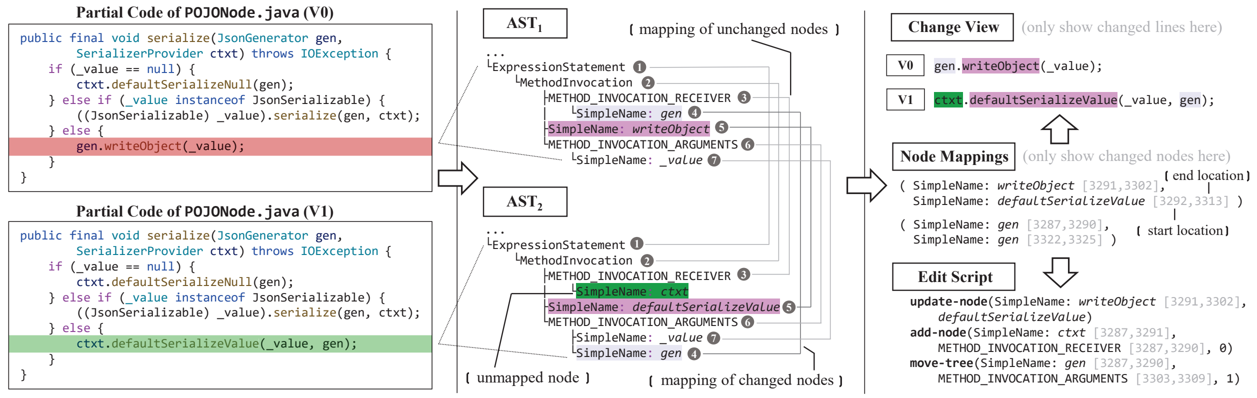
c) Edit Script (ES) Generation: This phase generates an ES that converts AST_1 to AST_2 through a sequence of edit actions [21]. These actions are derived from node mappings, including *node deletion* for unmapped nodes in AST_1, *node addition* for unmapped nodes in AST_2, *node update* for mapped nodes with changed values, and *tree move* for subtrees whose positions changed. As shown in Fig. 1, the generated edit script contains three actions: *update*, *add*, and *move*. The change view of diffs that colors code elements in the plain text can visualize different edit actions, as shown at the top of the right side of Fig. 1.

Following prior work [7], a perfect diff "reflects the commit author's intentions in the most accurate way". Operationally, given a diff $\mathcal{DG}$ generated by an ASTDiff technique and the oracle diff $\mathcal{DO}$ that was manually verified and refined, they consider $\mathcal{DG}$ as a *perfect diff* if it is identical to $\mathcal{DO}$, otherwise an *imperfect diff*. We further classify inaccurate mappings in an imperfect diff into three types. For a mapping $p = (n_1, n_2) \in \mathcal{DO}$, if n_1 and n_2 are both unmapped nodes in $\mathcal{DG}$, then we consider p a *missing mapping* for $\mathcal{DG}$. For a mapping $p = (n_1, n_2) \in \mathcal{DG}$, if n_1 and n_2 are both unmapped nodes in $\mathcal{DO}$, then we consider p an *arbitrary mapping*; otherwise, if n_1 and n_2 are not in the same mapping in $\mathcal{DO}$, then we consider p a *wrong mapping*.

B. Related Work

ASTDiff can be traced back to the algorithm proposed by Yang et al. [22], which recursively matches nodes between two ASTs using the Longest Common Subsequence (LCS) [23] algorithm. Since then, several ASTDiff approaches [2], [24]–[30] have been proposed over the past two decades. ChangeDistiller [2], [31] matches nodes based on string similarity and the proportion of common leaf nodes. However, it only operates on statement-level AST structures and cannot track fine-grained changes within statements.

GumTree [3] works on token-level AST (i.e., full AST) to produce fine-grained mappings, which has become the most widely used ASTDiff technique nowadays. It conducts two phases to match nodes: *top-down phase* matches identical subtrees based on isomorphism hash [32] which encodes the subtree's content and structure; *bottom-up phase* matches internal nodes according to common descendant proportion and performs *recovery matching* for newly mapped nodes to match their unmapped descendants via a tree differencing algorithm


Partial Code of POJONode.java (V0)

```
public final void serialize(JsonGenerator gen,
        SerializerProvider ctxt) throws IOException {
    if (_value == null) {
        ctxt.defaultSerializeNull(gen);
    } else if (_value instanceof JsonSerializable) {
        ((JsonSerializable) _value).serialize(gen, ctxt);
    } else {
        gen.writeObject(_value);
    }
}
```

Partial Code of POJONode.java (V1)

```
public final void serialize(JsonGenerator gen,
        SerializerProvider ctxt) throws IOException {
    if (_value == null) {
        ctxt.defaultSerializeNull(gen);
    } else if (_value instanceof JsonSerializable) {
        ((JsonSerializable) _value).serialize(gen, ctxt);
    } else {
        ctxt.defaultSerializeValue(_value, gen);
    }
}
```

AST_1

```
...
└ExpressionStatement ①
    └MethodInvocation ②
        ├METHOD_INVOCATION_RECEIVER ③
        │   └SimpleName: gen ④
        ├SimpleName: writeObject ⑤
        └METHOD_INVOCATION_ARGUMENTS ⑥
            └SimpleName: _value ⑦
```

AST_2

```
...
└ExpressionStatement ①
    └MethodInvocation ②
        ├METHOD_INVOCATION_RECEIVER ③
        │   └SimpleName: ctxt
        ├SimpleName: defaultSerializeValue ⑤
        └METHOD_INVOCATION_ARGUMENTS ⑥
            ├SimpleName: _value ⑦
            └SimpleName: gen ④
```

(mapping of unchanged nodes)
(unmapped node)
(mapping of changed nodes)
Change View
(only show changed lines here)
V0
V1
Node Mappings
(only show changed nodes here)
(end location)
(start location)
Edit Script

```
V0  gen.writeObject(_value);
V1  ctxt.defaultSerializeValue(_value, gen);
```

```
( SimpleName: writeObject [3291,3302],
  SimpleName: defaultSerializeValue [3292,3313] )
( SimpleName: gen [3287,3290],
  SimpleName: gen [3322,3325] )
```

```
update-node(SimpleName: writeObject [3291,3302],
    defaultSerializeValue)
add-node(SimpleName: ctxt [3287,3291],
    METHOD_INVOCATION_RECEIVER [3287,3290], 0)
move-tree(SimpleName: gen [3287,3290],
    METHOD_INVOCATION_ARGUMENTS [3303,3309], 1)
```

Fig. 1. ASTDiff for the modified file ”POJONode.java” of JacksonDatabind/97 in Defects4J.

RTED [33]. These strategies facilitate tracking changes within statements and produce concise edit scripts.

Researchers have improved GumTree on several aspects in recent years [4]–[7], [14], [16], [34]–[38]. MTDiff [5] refines GumTree's matching heuristics to optimize move actions and shorten the edit script. IJM [14] proposes *partial matching* that matches nodes in subtrees with different syntactic types and *name-aware matching* that matches nodes based on their names. ClDiff [35] groups edit actions according to the syntactic hierarchy and links them with semantic relationships. HyperDiff [39] aims to produce the same mappings as GumTree with less time cost by leveraging efficient data structures. GumTree-simple [4] replaces the costly RTED with a heuristic LCS-based recursive algorithm, which speeds up the node matching process by 50x-281x and produces smaller edit scripts. To further reduce the edit script size, DiffAutoTuning [16] traverses the parameter configurations of GumTree and executes them on a set of file pairs to obtain the smallest edit script. To improve the accuracy, iASTMapper [6] proposes an iterative node matching approach that calculates similarity scores of nodes and matches those with the best score in each iteration. RM-ASTDiff [7] uses several language clues such as refactoring and statement change patterns to match nodes, achieving the currently best accuracy among existing techniques.

III. Motivation

We conducted a preliminary investigation on DiffBenchmark against RM-ASTDiff (RMD), iASTMapper (IAM), DiffAutoTuning (DAT), GumTree-simple (GTS), and GumTree (GT). The key findings are that these baseline techniques produce a considerable number of inaccurate mappings, among which the same inaccurate mappings may be produced by different techniques, and even some inaccurate mappings caused by different reasons have certain similarities. Therefore, we summarize common observations of inaccurate mappings to facilitate their identification and fixing.

A. Observation #1: Type-related Inaccurate Mappings

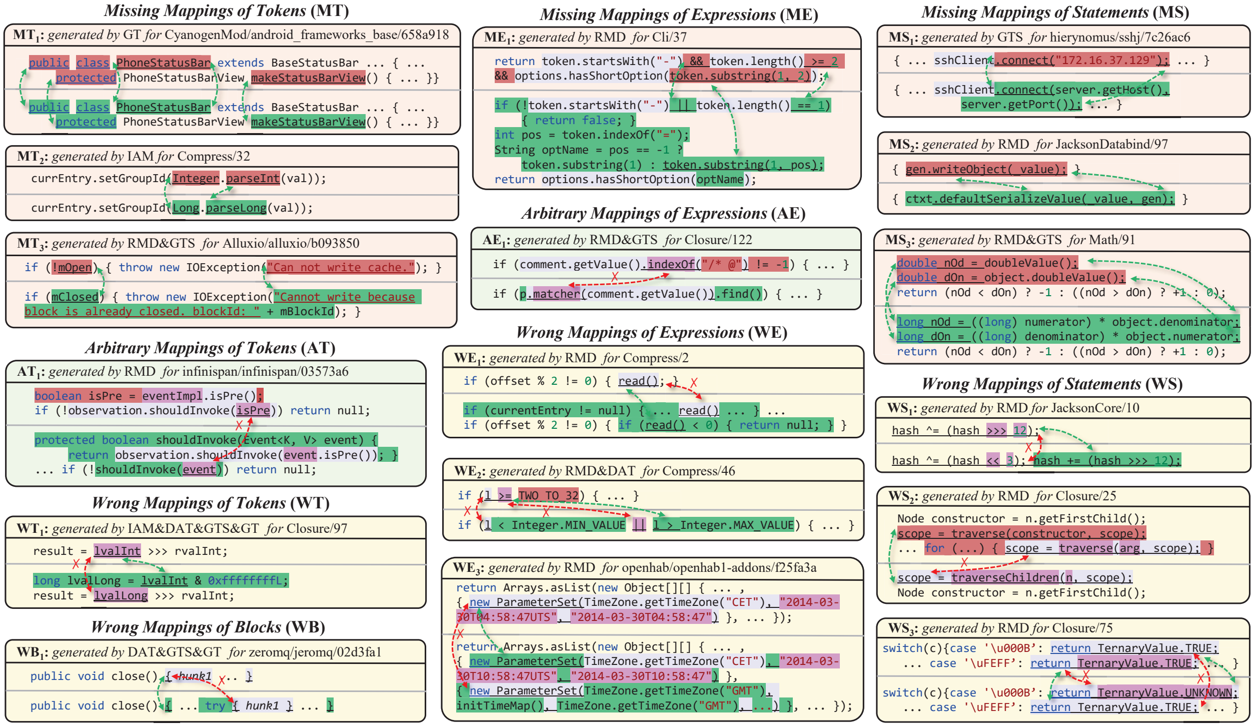
Our first observation is that inaccurate mappings mainly occur at certain node types, reflecting that existing tools often violate type-specific constraints when generating diffs. We classify common inaccurate mappings into nine categories based on their inaccuracy types (i.e., missing mapping, arbitrary mapping, and wrong mapping) and syntax levels (i.e., tokens, expressions, statements, and blocks). Fig. 2 illustrates the nine categories with real-world examples. Each example corresponds to a diff generated by the evaluated tools on a real-world commit in DiffBenchmark and points out its differences from the oracle diff. Note that most of the diffs are generated by RMD, since we are more interested in code changes that even the most accurate RMD cannot handle accurately.

The evaluated approaches are prone to mismatch syntax elements involving complex changes, such as variables ($\underline{AT_1}$ and $\underline{WT_1}$), method invocations ($\underline{ME_1}$ and $\underline{AE_1}$), infix expressions ($\underline{ME_1}$ and $\underline{WE_2}$), and statements ($\underline{MS_2}$ and $\underline{WS_2}$). For example, $\underline{AT_1}$ contains an arbitrary mapping of a deleted variable `isPre` that has been removed from a method invocation's arguments. Due to ignoring these constraints, all five tools arbitrarily match `isPre` and `event`. This example also involves moving a method invocation to a new method declaration, i.e., refactoring, and only the refactoring-aware approach RMD correctly matched it.

Therefore, to identify and fix these inaccurate mappings, type-specific constraints should be utilized. Specifically, we propose two constraints. The first constraint is that components under the mapped syntax elements should also be mapped. For example, missing mapped tokens in $\underline{MT_1}$ and $\underline{MT_2}$ and the two blocks of method bodies in $\underline{WB_1}$ all have unique roles according to their parents and should be mapped. The second constraint is that each syntax element should be mapped to its most relevant counterpart and should not be mapped to an irrelevant syntax element. For example, $\underline{AE_1}$ arbitrarily matched two distinct Java APIs, while $\underline{WE_2}$ failed to match the most relevant expression.

B. Observation #2: Context-related Inaccurate Mappings

Our second observation is that inaccurate mappings are often associated with four types of contexts of mapped nodes, as described below. These contexts help locate potentially inaccurate mappings and avoid traversing the whole AST.



Missing Mappings of Tokens (MT)

MT$_1$: *generated by* GT *for* CyanogenMod/android_frameworks_base/658a918

```
public class PhoneStatusBar extends BaseStatusBar ... { ...
    protected PhoneStatusBarView makeStatusBarView() { ... }}

public class PhoneStatusBar extends BaseStatusBar ... { ...
    protected PhoneStatusBarView makeStatusBarView() { ... }}
```

MT$_2$: *generated by* IAM *for* Compress/32

```
currEntry.setGroupId(Integer.parseInt(val));

currEntry.setGroupId(Long.parseLong(val));
```

MT$_3$: *generated by* RMD&GTS *for* Alluxio/alluxio/b093850

```
if (!mOpen) { throw new IOException("Can not write cache."); }

if (mClosed) { throw new IOException("Cannot write because
block is already closed. blockId: " + mBlockId); }
```

Arbitrary Mappings of Tokens (AT)

AT$_1$: *generated by* RMD *for* infinispan/infinispan/03573a6

```
boolean isPre = eventImpl.isPre();
if (!observation.shouldInvoke(isPre)) return null;

protected boolean shouldInvoke(Event<K, V> event) {
    return observation.shouldInvoke(event.isPre()); }
... if (!shouldInvoke(event)) return null;
```

Wrong Mappings of Tokens (WT)

WT$_1$: *generated by* IAM&DAT&GTS&GT *for* Closure/97

```
result = lvalInt >>> rvalInt;

long lvalLong = lvalInt & 0xffffffffL;
result = lvalLong >>> rvalInt;
```

Wrong Mappings of Blocks (WB)

WB$_1$: *generated by* DAT&GTS&GT *for* zeromq/jeromq/02d3fa1

```
public void close() { hunk1 ... }

public void close() { ... try { hunk1 } ... }
```

Missing Mappings of Expressions (ME)

ME$_1$: *generated by* RMD *for* Cli/37

```
return token.startsWith("-") && token.length() >= 2
&& options.hasShortOption(token.substring(1, 2));

if (!token.startsWith("-") || token.length() == 1)
    { return false; }
int pos = token.indexOf("=");
String optName = pos == -1 ?
    token.substring(1) : token.substring(1, pos);
return options.hasShortOption(optName);
```

Arbitrary Mappings of Expressions (AE)

AE$_1$: *generated by* RMD&GTS *for* Closure/122

```
if (comment.getValue().indexOf("/* @") != -1) { ... }

if (p.matcher(comment.getValue()).find()) { ... }
```

Wrong Mappings of Expressions (WE)

WE$_1$: *generated by* RMD *for* Compress/2

```
if (offset % 2 != 0) { read(); }

if (currentEntry != null) { ... read() ... } ...
if (offset % 2 != 0) { if (read() < 0) { return null; } }
```

WE$_2$: *generated by* RMD&DAT *for* Compress/46

```
if (l >= TWO_TO_32) { ... }

if (l < Integer.MIN_VALUE || l > Integer.MAX_VALUE) { ... }
```

WE$_3$: *generated by* RMD *for* openhab/openhab1-addons/f25fa3a

```
return Arrays.asList(new Object[][] { ... ,
{ new ParameterSet(TimeZone.getTimeZone("CET"), "2014-03-
30T04:58:47UTS", "2014-03-30T04:58:47") }, ... });

return Arrays.asList(new Object[][] { ... ,
{ new ParameterSet(TimeZone.getTimeZone("CET"), "2014-03-
30T10:58:47UTS", "2014-03-30T10:58:47") },
{ new ParameterSet(TimeZone.getTimeZone("GMT"),
initTimeMap(), TimeZone.getTimeZone("GMT"), ...) }, ... });
```

Missing Mappings of Statements (MS)

MS$_1$: *generated by* GTS *for* hierynomus/sshj/7c26ac6

```
{ ... sshClient.connect("172.16.37.129"); ... }

{ ... sshClient.connect(server.getHost(),
          server.getPort()); ... }
```

MS$_2$: *generated by* RMD *for* JacksonDatabind/97

```
{ gen.writeObject(_value); }

{ ctxt.defaultSerializeValue(_value, gen); }
```

MS$_3$: *generated by* RMD&GTS *for* Math/91

```
double nOd = doubleValue();
double dOn = object.doubleValue();
return (nOd < dOn) ? -1 : ((nOd > dOn) ? +1 : 0);

long nOd = ((long) numerator) * object.denominator;
long dOn = ((long) denominator) * object.numerator;
return (nOd < dOn) ? -1 : ((nOd > dOn) ? +1 : 0);
```

Wrong Mappings of Statements (WS)

WS$_1$: *generated by* RMD *for* JacksonCore/10

```
hash ^= (hash >>> 12);

hash ^= (hash << 3); hash += (hash >>> 12);
```

WS$_2$: *generated by* RMD *for* Closure/25

```
Node constructor = n.getFirstChild();
scope = traverse(constructor, scope);
... for (...) { scope = traverse(arg, scope); }

scope = traverseChildren(n, scope);
Node constructor = n.getFirstChild();
```

WS$_3$: *generated by* RMD *for* Closure/75

```
switch(c){case '\u000B': return TernaryValue.TRUE;
  ... case '\uFEFF': return TernaryValue.TRUE; ... }

switch(c){case '\u000B': return TernaryValue.UNKNOWN;
  ... case '\uFEFF': return TernaryValue.TRUE; ... }
```

(changed code marked as deleted/added/updated/moved; inaccuracy-related code marked as deleted/added/updated/moved)

Fig. 2. Examples of Inaccurate Mappings Generated by ASTDiff Techniques for Real-world Commits. These tools include: RM-ASTDiff (RMD), iASTMapper (IAM), DiffAutoTuning (DAT), GumTree-simple (GTS), and GumTree (GT). Green arrows point to the missing mapped code elements; red arrows point to the incorrect mapped code elements.

Parent context refers to the situation that the mapped parent nodes contain mis-mapped or missing-mapped child nodes (e.g. MT$_1$, MS$_1$, and WB$_1$). For example, MT$_1$ shows that GT skipped matching leaf children of large non-identical subtrees (code changes in method bodies are omitted here). This context can be used to identify inaccurate mappings of child nodes based on the mapping status of their parent nodes.

Children context refers to the situation that the parent nodes of the mapped child nodes are not accurately mapped (e.g. ME$_1$, WE$_2$, and WE$_3$). We use this context to identify inaccurate mappings of parent nodes based on the mapping status of their child nodes.

Inner context refers to the situation that nodes within the same block are not accurately mapped (e.g. MT$_3$, ME$_1$, and WE$_1$). In some cases, inaccurate mappings involve nodes moved to added code chunks or nodes moved out of deleted code chunks, which cannot be detected using the parent or children context. We observed that most of the changed nodes, especially tokens and expressions, still reside in the block where the change occurs (i.e., locality). For example, in ME$_1$, the method invocation `token.substring(1, 2)` is moved and updated, but still within the same block.

Nearby context refers to the situation that statements around the mapped statements are not accurately mapped (e.g. MS$_3$, WS$_1$, and WS$_2$). In cases where inaccurate statement mappings are more complicated and even span blocks, we observe that statements around mapped statements are more likely to be mapped (i.e., transitivity). For example, in WS$_2$, the statement in the first line has a nearby mis-mapped statement in the second line, which has been moved and updated.

C. Observation #3: Coexistence of Inaccurate Mappings

The examples in Fig. 2 only contain one category of inaccurate mappings for illustration purposes. However, an imperfect diff may contain multiple categories of inaccurate mappings, and even one category of inaccurate mappings may occur multiple times in an imperfect diff (e.g. ME$_1$, MS$_3$, WE$_2$, and WS$_3$). What's worse, different inaccurate mappings can be associated and interact with each other, further hindering their identification.

Based on the above observations, we propose that: 1) different contexts should be used to identify potential inaccurate mappings(reduce false negatives); 2) type-specific constraints should be used to confirm and fix them (reduce false positives); and 3) an iterative and incremental approach should be developed to fix various types of inaccurate mappings and improve the context to reveal hidden inaccurate mappings.

IV. Approach

In this section, we present our approach DiffFix with an overview and key technical treatments.

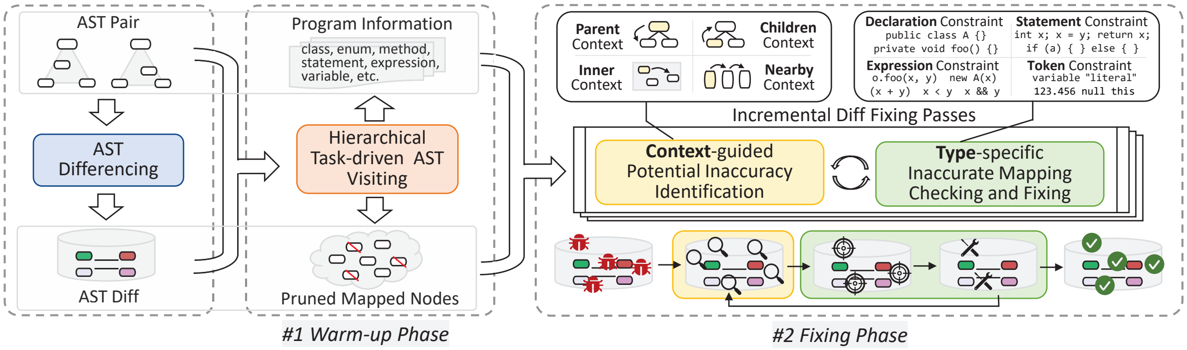


Fig. 3. Overview of DIFFFIX framework

A. Overview of DIFFFIX

Fig. 3 illustrates an overview of DIFFFIX, which contains a warm-up phase to perform initializations and a fixing phase to perform diff fixing. DIFFFIX takes as input an AST pair and a diff generated by an ASTDiff tool on the AST pair. Then, the *warm-up phase* collects type-specific program information and generates pruned mapped node sets by traversing each AST using a hierarchical task-driven visitor. After that, the *fixing phase* incrementally refines the AST diff through nine passes. Each pass traverses a pruned mapped node set to identify potentially inaccurate mappings using one of the four contexts mentioned above, then leverages program information pre-fetched in the warm-up phase to confirm and fix them.

B. The Warm-up Phase

This phase serves two purposes: 1) to provide the initial environment for the fixing phase by conducting basic checks to fix inaccurate mappings and obtaining the global information used by the fixing phase; 2) to improve the efficiency of the fixing phase by pruning the traversal node set and pre-fetching the local information required by the fixing phase.

The following three measures are used to establish the initial environment: i) *Node type checking* removes mappings in the original mappings where the two nodes are of different types. ii) *Identical mapping checking* checks whether the descendants of two mapped identical nodes are mapped correspondingly and fixes inconsistencies. iii) *Global information collection* maintains global information in each AST to reduce overhead for subsequent analysis in the fixing phase. Specifically, we record the previous and next statement in the pre-order traversal sequence of the method body, identical statements, renamed methods, and unmapped nodes in each AST.

The following three optimization strategies are employed to enhance the efficiency of the fixing phase: i) *Local information collection (opt_1)* maintains some local information (such as method invocations contained in nested blocks) to avoid redundant traversal of nodes during the fixing phase. ii) *Declaration-level pruning (opt_2)* removes the descendants of identical mapped declarations (e.g., classes, methods, and enums) from the set of all mapped AST_1 nodes, because they are already optimal after identical mapping checking and thus do not require additional fixes. iii) *Statement-level pruning (opt_3)* further removes the descendants of identical mapped statements from the pruned node set, to reduce the nodes requiring analysis.

Although the warm-up phase integrates various functionalities, it only needs to traverse AST_1 and AST_2 once. To achieve this, we design a hierarchical task-driven AST visitor that top-down traverses classes, methods, and statements with their customized *information collection* and *pruning* task executors. When traversing AST_1, it also performs *mapping type checking* and *identical mapping checking* on mapped nodes, and marks the subtree in AST_2 identical to the currently traversed one. When traversing AST_2, it performs *information collection* and skips the marked subtrees.

C. The Fixing Phase

The fixing phase consists of nine fixing passes that are executed sequentially. In designing the passes, we attempt to leverage all four types of context as effectively as possible. Table I lists these passes in order, where each identifies potential inaccurate mappings based on a specific *context* and further checks and fixes them based on *type* constraints.

We deliberately considered the interdependencies when ordering the passes. Specifically, $MatchUnique$ is executed first to provide a more complete context for subsequent passes. The second to fifth passes are used to fix arbitrary and wrong mappings, providing a more accurate context for the subsequent four passes that target missing mappings. $FixChildren$ is executed after $FixParent$ to ensure that checks are done on a refined parent context. $FixInner$ and $FixNearby$ are then executed to check for inaccuracies that are more difficult to identify. Then $MatchInner$ and $MatchNearby$ are executed to find some non-local missing mappings, which may provide some new mapped nodes for $MatchParent$. $MatchChildren$ is executed last to ensure that the newly mapped parent nodes can be checked.

Each fixing pass identifies and fixes inaccuracies by traversing one of two pruned node subsets $\mathcal{P}$ and $\mathcal{P}'$ produced by the warm-up phase. $\mathcal{P}$ excludes all descendants of mapped identical declarations and statements; $\mathcal{P}'$ retains the root nodes of mapped identical statements in changed declarations. Most fixing passes can use $\mathcal{P}$ to satisfy node analysis without affecting fixing ability; $FixNearby$, $MatchNearby$,

TABLE I
OVERVIEW OF FIXING PASSES

Name	Context	Target	Type*	Example
MatchUnique	Parent Context	missing mappings in unique unmapped child nodes	D; S	MS$_1$, MS$_2$
FixParent	Children Context	arbitrary/wrong mappings in parent nodes	E; T	AT$_1$, WE$_3$
FixChildren	Parent Context	arbitrary/wrong mappings in child nodes	S; E; T	WB$_1$, WE$_2$
FixInner	Inner Context	arbitrary mappings of nodes within the same block	S; E; T	AE$_1$, WE$_1$
FixNearby	Nearby Context	arbitrary/wrong mappings in adjacent statements	S	WS$_1$, WS$_2$, WS$_3$, WT$_1$
MatchInner	Inner Context	missing mappings of nodes within the same block	S; E; T	MT$_3$, ME$_1$
MatchNearby	Nearby Context	missing mappings in adjacent statements	S	MS$_3$
MatchParent	Children Context	missing mappings in parent nodes	S; E; T	ME$_1$
MatchChildren	Parent Context	missing mappings in child nodes	S; E; T	MT$_1$, MT$_2$, ME$_1$

*The mainly used type constraints, including declaration (D), statement (S), expression (E), and token (T) constraints.

and $MatchParent$ need $\mathcal{P}'$ to check adjacent statements or parents through root nodes of identical statements. When a mapping changes, $\mathcal{P}$ and $\mathcal{P}'$ are also updated synchronously.

Next, we will elaborate on the design of each fixing pass.

1) $MatchUnique$: The pass is based on the observation that if two nodes form a unique unmapped pair of a certain type within their mapped parents, then the two nodes are likely to be mapped. For example, the unmapped statements in MS$_1$ and MS$_2$ are all the unique unmapped statements in the corresponding block and have the same type and common elements, so they are identified as missing mappings by $MatchUnique$.

Algorithm 1 illustrates the process of $MatchUnique$. First, it traverses through m_1 in $\mathcal{P}$ that have a corresponding mapped node m_2. If m_1 and m_2 are composite statements, it retrieves and pairs unmapped child nodes of the same type from both m_1 and m_2 ($uniqueUnmappedChildrenPairs$). Then, it checks whether there are common elements for each pair of unique unmapped children based on the syntax type, such as the left value of an assignment statement and the test condition of an `if` statement, and tries to match identified missing mappings ($tryToMatchInComposite$). If m_1 and m_2 are class declarations, a similar process is conducted.

Algorithm 1: MatchUnique

```
   Data: Pruned mapped $AST_1$ nodes $\mathcal{P}$
1  foreach $m_1 \in \mathcal{P}$ do
2      if $isMapped(m_1)$ then
3          $m_2 \leftarrow getMapped(m_1)$
4          if $isCompositeStatement(m_1)$ then
5              $\mathcal{U} \leftarrow uniqueUnmappedChildrenPairs(m_1, m_2)$
6              foreach $(u_1, u_2) \in \mathcal{U}$ do
7                  $tryToMatchInComposite(u_1, u_2)$
8              end
9          end
10         else if $isClassDeclaration(m_1)$ then
11             $\mathcal{D} \leftarrow uniqueUnmappedChildrenPairs(m_1, m_2)$
12             foreach $(d_1, d_2) \in \mathcal{D}$ do
13                 $tryToMatchInClass(d_1, d_2)$
14             end
15         end
16     end
17 end
```

2) $FixParent$: This pass first executes the same process as lines 1 to 3 of Algorithm 1, and then conducts two checks for parent nodes of mapped m_1 and m_2. i) If a pair of mapped nodes has the same parent types but their parents are not mapped with each other, there may exist a suboptimal mapping (such as the class instance creations in WE$_3$). $FixParent$ identifies and replaces such suboptimal mapping with a pair of parent nodes which has more mapped elements. ii) If a pair of mapped nodes has different parent types, there may exist inaccurate mappings related to code deletion and addition (such as the deleted variable `isPre` in AT$_1$). $FixParent$ removes such arbitrary mappings of already deleted or newly created variables.

3) $FixChildren$: This pass has a similar process as $FixParent$ but checks for mis-mapped child nodes that have unique roles to their parent nodes (such as blocks in WB$_1$). It can also fix some inaccuracies that are partially solved by $FixParent$. In WE$_2$, $FixParent$ identifies that `l >= TWO_TO_32` is wrongly mapped with a logical expression by checking their operator compatibility and rematches it with `l < Integer.MIN_VALUE`; subsequently, $FixChildren$ identifies its better partner `l > Integer.MAX_VALUE` in the test conditions of mapped `if` statements.

4) $FixInner$: This pass first removes a type of arbitrary mappings that contain a solely mapped node in added or deleted blocks (such as the mapped `read()` in WE$_1$), then removes arbitrary mappings of method invocations that are incompatible in method name or return value. For example, in AE$_1$, one method invocation invokes `indexOf()`, which typically returns an `int` value, and the other returns an `Object` since it is the receiver of another method invocation. $FixInner$ uses several heuristics to infer return values and identify such inaccuracies.

5) $FixNearby$: This pass uses the mapped statement as an anchor to check for mismatches of the statement itself or nearby statements. Since newly mapped statements can serve as new anchors, it checks statements in multiple iterations, as illustrated in Algorithm 2. First, $\mathcal{S}$ is initialized as the set of all leaf statements in $\mathcal{P}'$. If $\mathcal{S}$ is not empty, $FixNearbyRound$ is invoked to execute an iteration, and $\mathcal{S}$ is updated to the set of newly mapped statements. Then, the next iteration takes place until $\mathcal{S}$ is empty.

In $FixNearbyRound$, four subprocesses are executed to check for different types of inaccurate statement mappings:

a) $checkNearbyBetter$: This process seeks a better mapping which has more common elements than s_1 and s_2 in four pairs of statements: ($prev_1$, s_2), ($next_1$, s_2), (s_1,

Algorithm 2: FixNearby

```
   Data: Pruned mapped $AST_1$ nodes $\mathcal{P}'$
1  Function FixNearbyRound($\mathcal{S}$):
2      $\mathcal{R} \leftarrow \varnothing$
3      foreach $s_1 \in \mathcal{S}$ do
4          if $isMapped(s_1)$ then
5              $s_2 \leftarrow getMapped(s_1)$
6              if $checkNearbyBetter(s_1, s_2, R)$ then
7                  continue
8              end
9              $checkNearbyWM(s_1, s_2, \mathcal{R})$
10             $checkIdenticalWM(s_1, s_2, \mathcal{R})$
11             $checkNearbyInnerStmtWM(s_1, s_2)$
12         end
13     end
14     return $\mathcal{R}$
15 $\mathcal{S} \leftarrow getAllLeafStatements(\mathcal{P}')$
16 while $\mathcal{S} \neq \varnothing$ do
17     $\mathcal{S} \leftarrow FixNearbyRound(\mathcal{S})$
18 end
```

$prev_2$), and (s_1, $next_2$), which $prev_1$/$prev_2$ are the previous statements of s_1/s_2 and $next_1$/$next_2$ are the next statements of s_1/s_2. For example, in WS$_1$, s_1 and $next_2$ are better than s_1 and s_2 because they have common right operands. If a better mapping is found, it replaces the original mapping with the better one, updates $\mathcal{R}$, and skips subsequent processes.

b) $checkNearbyWM$: This process examines mismatches in nearby statements of s_1 and s_2 and seeks better mappings in four pairs of statements: ($prev_1$, $prev_2$), ($next_1$, $next_2$), ($prev_1$, $next_2$), and ($next_1$, $prev_2$). For example, in WS$_2$, the mapping of $next_1$ and $prev_2$ is more appropriate than the original mapping of $prev_2$ and a statement within a deleted block. If a better mapping is found, it replaces the original mapping with the better one and updates $\mathcal{R}$.

c) $checkIdenticalWM$: If s_1 and s_2 are identical statements and have not been examined before, this process examines all statements in AST_1 and AST_2 identical to them and fixes mismatches. If one identical statement exists in an AST but multiple identical statements exist in the opposite (WS$_3$), it chooses the best partner for the former from the latter candidates based on the mapping status of their parents and siblings. If both AST_1 and AST_2 have multiple identical statements, it sorts these statements by position and sequentially compares them to the original mappings to determine the superior.

d) $checkNearbyInnerStmtWM$: Since code moves across nearby statements are common in practice, this process fixes mis-mapped moved nodes in nearby statements. For example, in WT$_1$, the variable `lvalInt` moved to the previous statement is inaccurately mapped with a newly defined variable `lvalLong`. $FixNearby$ leverages variable definition and use information to check such inaccuracies.

6) $MatchInner$: This pass matches missing mapped nodes within mapped blocks according to their types, which mainly include return statement, infix expression, method invocation (ME$_1$) and string literal (MT$_3$). Specifically, $MatchInner$ performs two checks to match missing mapped method invocations in two mapped blocks b_1 and b_2: i) If there exists only one pair of unmapped identical method invocations i_1 in b_1 and i_2 in b_2, and at most one block is nested within b_1 and b_2 (to avoid mapping that spans multiple blocks), then it matches i_1 with i_2. ii) Otherwise, it obtains the candidate sets $\mathcal{I}_1$ and $\mathcal{I}_2$ with unmapped method invocations in b_1 and b_2 respectively that have no internal mappings but have adjacent mapped nodes. For each i_1 in $\mathcal{I}_1$, it traverses $\mathcal{I}_2$ to find i_2 that has the maximum number (at least one) of common elements (receiver, name, and arguments) with i_1. If there are multiple candidates, it selects the one as i_2 with the most similar name to i_1. If i_2 exists, it traverses $\mathcal{I}_1$ to check whether the best candidate for i_2 is i_1. If i_1 and i_2 are mutually the best candidates, then they should be mapped.

7) $MatchNearby$: To leverage newly mapped statements, this pass performs multiple iterations similar to lines 12 to 14 of Algorithm 2, but in each iteration identifies missing mapped nearby statements of s_1 and s_2 in $\mathcal{S}$. For example, in MS$_3$, the first iteration matches the variable declarations of `dOn` previous to the mapped `return` statements, and the second iteration matches the variable declarations of `nOd` previous to the newly mapped statements. DIFFFIX determines whether two statements can be mapped by checking key features specific to their types (such as the declared variable in a variable declaration), rather than by similarity, thus alleviating the problem of missing statements with large changes.

8) $MatchParent$: This pass uses mapped nodes as anchors to identify their missing mapped parents, which also performs multiple iterations similar to Algorithm 2 but directly uses $\mathcal{P}'$ as the initial node set. In each iteration, it conducts three checks: i) first, it matches unmapped parents of the same type if their main elements are mapped, such as the infix expressions in ME$_1$ with mapped left operands; ii) second, it checks parents of different types to identify missing mappings of parents with moved elements such as `exp.foo()` and `var=expr, var.foo()`; iii) finally, it checks unmapped pairs of parent and grandparent of the same type to handle nested expressions such as `var+1` and `(var)+1`.

9) $MatchChildren$: This pass identifies missing mapped children within mapped nodes according to their types, such as class declarations (MT$_1$) and method invocations (MT$_2$). It also complements $MatchParent$ to fix the remaining missing mappings. For example, in ME$_1$, the infix expressions `token.length() >= 2` and `token.length() == 1` are mapped in $MatchParent$, and then $MatchChildren$ matches the unmapped operators and right operands.

V. EVALUATION

In this section, we report on a series of experiments that are designed to evaluate the effectiveness of DIFFFIX.

A. Research Questions

Through the experiments, we attempt to answer the following research questions:

- *RQ1:* Can DIFFFIX effectively improve node mapping accuracy and produce more perfect diffs?
- *RQ2:* What is the time cost of DIFFFIX? Can optimizations make DIFFFIX more efficient?

- *RQ3:* How does DIFFFIX affect edit script size?
- *RQ4:* What are the execution times and contributions of different subprocesses in DIFFFIX?

B. Experimental Setup

1) Dataset: To the best of our knowledge, DiffBenchmark [7] is the only available benchmark of human-verified AST diffs (i.e., oracle diffs), so we used it to evaluate DIFFFIX's improvements on node mapping accuracy. It includes two datasets: Defects4J comprises 800 bug-fixing commits from 17 open-source projects [40]; RefOracle comprises 188 refactoring commits from 105 open-source projects [41]. Since one commit[1] from RefOracle does not have the oracle diff, we used a total of 987 commits from DiffBenchmark[2] for evaluation.

2) Subjects: We applied DIFFFIX to five ASTDiff techniques to evaluate its effectiveness: RMD, IAM, DAT, GTS, and GT. The first four are all state-of-the-art techniques proposed in the past two years, and GT remains well-maintained and most widely used. These techniques all use the same AST structure and diff format as DiffBenchmark, allowing us to evaluate their node mapping accuracy. IJM and MTDiff were excluded because the AST structure they used is inconsistent with DiffBenchmark. Additionally, their node matching heuristics have some defects that have been discussed [7], while RMD and IAM can be regarded as improvements on them.

We adopted the original implementations of RMD, GTS, and GT, and adapt IAM and DAT based on the original implementations to support the evaluation. The original implementation of IAM ignores syntax structures such as `Javadoc` and `Varargs` and produces some type-inconsistent mappings during node matching, thus generating many trivial inaccuracies. Therefore, we post-process IAM mappings to supplement missing mappings of `Javadoc` and `Varargs` related nodes, and remove type-inconsistent mappings to better reflect non-trivial inaccuracies. DAT requires a time-consuming preprocessing phase to search for the best GumTree parameter setting which may not be unique. To ensure consistency across repeated experiments, we applied the preprocessing phase to each file pair in the dataset and saved the setting that had the least execution time. When evaluating DAT, we used the saved setting without additional preprocessing.

3) Experimental procedure: We performed ASTDiff on commits from DiffBenchmark using five techniques with DIFFFIX applied, and compared the results to those without DIFFFIX applied by various metrics. For time-related metrics, we repeated the experiments 10 times and calculated the average values. Experiments were conducted on Windows 10 22H2 with Intel Core i7-10700 CPU and 48.0 GB RAM. To facilitate reproducibility and future research, we provided the source code of DIFFFIX and experimental data at https://github.com/guofeng99/DiffFix.

Due to the space limitation, the following sections show the results on Defects4J. The overall trend of the results on RefOracle (included in the above link) is similar to that of Defects4J, which also illustrates the superiority of DIFFFIX in terms of accuracy and efficiency.

C. Effectiveness (RQ1)

The effectiveness of DIFFFIX is reflected in its improvement on the node mapping accuracy of existing techniques. We use the perfect diff rate (PDR) and the number of three inaccurate mappings to evaluate node mapping accuracy. PDR measures the proportion of generated diffs that are identical to oracle diffs [7], and has two variants: PDR_{token} checks all mappings, while PDR_{stmt} ignores inner-statement mappings. Table II shows the corresponding metrics' results in the second to sixth columns.

From Table II, we observed that: i) DIFFFIX improved PDR_{token} and PDR_{stmt} on all techniques, achieving the best PDR_{token} of 95.38% and the best PDR_{stmt} of 97.62% when applied to RMD; ii) DIFFFIX reduced missing mappings and wrong mappings on all techniques, but slightly increased arbitrary mappings (of some subtle method invocations) on two techniques. In addition, DIFFFIX showed a significant improvement on PDR_{token} on GT, because a large number of imperfect diffs of GT were caused by missing token mappings in declaration, which can be effectively handled by $MatchChildren$.

We mainly focus on the inaccurate mappings of RMD, because it can prevent most of the inaccurate mappings generated by other techniques. As a result, there are fewer remaining inaccurate mappings on RMD than other techniques. These remaining inaccuracies are mainly related to multi-mappings (i.e., one-to-many or many-to-one node mappings) which can only be captured by RMD among existing techniques. The process in RMD to detect multi-mappings is refactoring-based and quite heavyweight, so they are not handled by DIFFFIX.

To evaluate the effectiveness of DIFFFIX beyond DiffBenchmark, we selected 10,711 commits from 116 popular GitHub open source projects and evaluated whether DIFFFIX can effectively fix the diffs generated by RMD for these commits. We first obtained the first 200 Java projects with the most stars on GitHub before April 16, 2025, and filtered out projects that are only tutorials and resources or have no commits in the past four years, and finally retained 116 projects. For each project, we randomly selected 100 commits that only change a single Java file (for the convenience of manual verification) and contain both line additions and deletions (to ensure the basic difficulty of ASTDiff). If there are fewer than 100 commits that meet the conditions in a project, we select all of them. Finally, a total of 10,711 commits are obtained.

We used RMD to process these commits, of which 2 failed and 102 timed out (60 minutes). Among the remaining 10,607 diffs generated by RMD, DIFFFIX modified 1,688 (15.91%) of them. These modified diffs come from commits in 115 projects, which shows that DIFFFIX can affect a wide range of diffs. We randomly selected 320 of the 1,688 diffs and manually verified whether DIFFFIX can accurately fix inaccurate mappings. For each case, we compared the original diff

[1] https://github.com/glyptodon/guacamole-client/commit/ebb483
[2] https://github.com/pouryafard75/DiffBenchmark

TABLE II
THE EFFECT OF DIFFFIX ON ASTDIFF TECHNIQUES. THE DIFFERENCE IS GIVEN IN PARENTHESES, WHERE +/– INDICATE THE CHANGE VALUE, AND ↑/↓ INDICATE THE CHANGE RATE.

	PDR_{stmt} (%)	PDR_{token} (%)	Missing Mappings (#)	Arbitrary Mappings (#)	Wrong Mappings (#)	T_{diff} (ms)	ES Size (#)
RMD	89.38	86.00	719	81	251	34,108.1	39,954
RMD+DIFFFIX	97.62 (+8.25)	95.38 (+9.38)	164 (–555)	81 (+0)	42 (–209)	34,769.5 (↑1.94%)	38,871 (↓2.71%)
IAM	90.38	71.00	570	384	408	96,568.5	39,094
IAM+DIFFFIX	92.25 (+1.88)	80.75 (+9.75)	353 (–217)	400 (+16)	372 (–36)	97,158.0 (↑0.61%)	38,642 (↓1.16%)
DAT	82.00	70.62	177	1,279	986	32,472.9	37,238
DAT+DIFFFIX	86.50 (+4.50)	75.88 (+5.25)	174 (–3)	1,162 (–117)	787 (–199)	33,038.4 (↑1.74%)	37,407 (↑0.45%)
GTS	72.25	62.62	1,424	356	1,006	2,206.3	40,936
GTS+DIFFFIX	82.88 (+10.62)	72.25 (+9.62)	759 (–665)	362 (+6)	801 (–205)	2,757.1 (↑24.96%)	39,544 (↓3.40%)
GT	75.25	17.25	3,374	717	1,035	370,413.4	44,835
GT+DIFFFIX	82.88 (+7.62)	68.38 (+51.12)	892 (–2,482)	637 (–80)	831 (–204)	370,995.9 (↑0.16%)	39,903 (↓11.00%)

generated by RMD with the diff modified by DIFFFIX using the side-by-side change view provided by RMD, and recorded the inaccurate mapping categories contained in the original diff and three situations made by DIFFFIX: 1) accurately fixed; 2) not fixed; 3) introduced new inaccuracies.

The results of our manual verification show that DIFFFIX accurately fixed 298 of the 320 diffs, with a precision of 93.12%. The top five inaccurate mapping categories with the most accurately fixed instances are *missing mappings of statements* (#181), *missing mappings of expressions* (#51), *missing mappings of declarations* (#26), *wrong mappings of statements* (#24), and *arbitrary mappings of expressions* (#11). For instances that are not handled accurately, the most common categories are unfixed *wrong multi-mappings* (#4) and newly introduced *arbitrary mappings of statements* (#9). The latter is mainly caused by matching unrelated method invocation statements in the same context, such as `cb.onLoadFailed(e);` and `handleException(e);` in the same `catch` block[3].

Answer to RQ1: DIFFFIX effectively improved node mapping accuracy on different ASTDiff techniques. When applied to RMD, it achieved a perfect diff rate of 95.8%, demonstrating high precision and generalizability.

D. Efficiency (RQ2)

To measure the efficiency, we compared the node matching time of different techniques (i.e., ignoring the time for AST parsing and edit script generation) with the running time of DIFFFIX. The seventh column of Table II presents the T_{diff} across different configurations, which is the total time spent on node matching plus the optional time spent on diff fixing. As seen in Table II, DIFFFIX has a negligible overhead of less than 2% on four techniques, and takes less than a quarter of the time of the most efficient GTS.

As shown in Fig. 4, the running time of DIFFFIX on each diff has a similar distribution across different techniques. Most diffs are processed in less than 1 millisecond, with an average of 0.69 to 0.83 milliseconds. The maximum across different techniques is between 9.4 and 13.0 milliseconds.

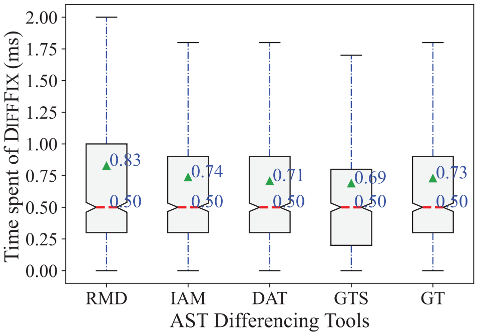


Fig. 4. The Boxplot of Time Spent of DIFFFIX per Diff

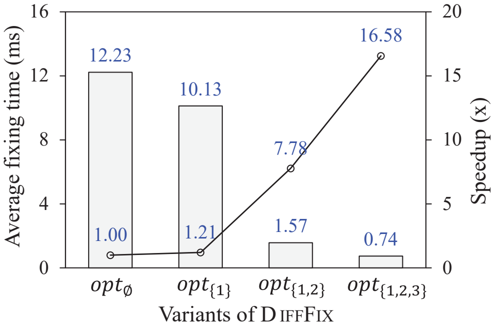


Fig. 5. The effect of optimizations of DIFFFIX.

The high efficiency of DIFFFIX is mainly brought by the three optimizations in the warm-up phase. To evaluate the impact of them on efficiency, we executed four DIFFFIX variants: $opt_{\emptyset}$ (without optimizations), $opt_{\{1\}}$ (with opt_1), $opt_{\{1,2\}}$ (with opt_1 and opt_2), and $opt_{\{1,2,3\}}$ (with opt_1, opt_2 and opt_3, which is the default setting). The fixing time and speedup (compared to $opt_{\emptyset}$) of variants on different techniques are close, so we further calculated the average value across different techniques. As seen from Fig. 5, all three optimizations have an incremental improvement on DIFFFIX's efficiency, ultimately achieving a 16.58x speedup. Among

[3] https://github.com/bumptech/glide/commit/50997b0

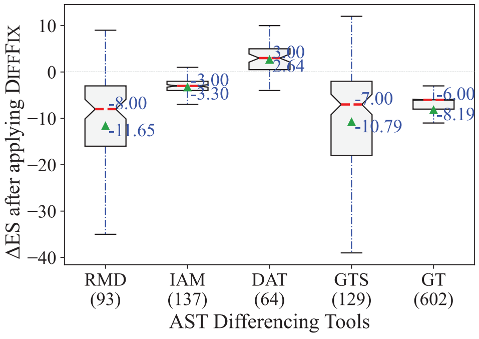


Fig. 6. The Boxplot of ΔES of DiffFix.

them, the pruning strategies (opt_2 and opt_3) have the greatest impact on DiffFix efficiency.

Answer to RQ2: DiffFix demonstrated high efficiency with negligible time cost on most techniques. The optimizations significantly improved its efficiency.

E. Effect on Edit Script Size (RQ3)

DiffFix changes node mappings and therefore affects the edit script. The last column of Table II shows the change in the total edit script size (ES size) after applying DiffFix to different techniques. DiffFix reduced ES size on four techniques, most on GT, and slightly increased ES size on DAT. The degree of reduction in ES size is mainly related to the number of reduced missing mappings. DiffFix reduced more arbitrary mappings on DAT than missing mappings, thus increasing node additions and deletions.

When DiffFix processes a diff, it may cause the corresponding ES size to increase, decrease, or remain the same. Fig. 6 shows the distribution of ES size change (ΔES) of the diffs whose ES size is changed by DiffFix, and the number of these diffs is given in parentheses. On the four techniques except DAT, most of the diffs changed by DiffFix reduced ES size, and the reduction was greater for RMD and GTS. On DAT, most of the changed diffs slightly increased ES size.

Inaccurate mappings inevitably lead to inaccurate edit scripts. For example, missing mappings derive deletions or insertions for unchanged, moved, or updated code, obscuring related code fragments. Arbitrary mappings derive moves and updates for unrelated deleted/added code fragments, introducing spurious changes. Similarly, wrong mappings inject noise into the edit scripts. These consequences negatively impact downstream tasks that typically rely on edit scripts.

Answer to RQ3: DiffFix reduced ES size on most techniques primarily by fixing missing mappings, while also alleviating the problem of arbitrarily matching nodes.

F. Impact of Subprocesses (RQ4)

The subprocesses in DiffFix include the warm-up phase and nine fixing passes in the fixing phase: $MatchUnique$ (MU), $FixParent$ (FP), $FixChildren$ (FC), $FixInner$ (FI), $FixNearby$ (FN), $MatchInner$ (MI), $MatchNearby$ (MN), $MatchParent$ (MP), and $MatchChildren$ (MC). Table III shows the impact of these subprocesses in terms of effectiveness and time cost, where **bold** indicates the maximum value in different algorithms, and underlined indicates the top three maximum values in the fixing passes.

TABLE III
Impact of Subprocesses

(a) The number of diffs that subprocesses take effect on (#)

	Warm-up Phase	Fixing Passes								
		MU	FP	FC	FI	FN	MI	MN	MP	MC
RMD	8	**45**	4	7	7	11	22	16	20	12
IAM	**30**	4	7	7	3	20	**38**	1	26	75
DAT	0	4	**10**	56	**33**	23	1	2	4	10
GTS	0	41	1	**71**	4	26	25	**30**	**29**	7
GT	0	26	4	70	23	**27**	3	12	20	**681**

(b) Total time spent of subprocesses (ms)

	Warm-up Phase	Fixing Passes								
		MU	FP	FC	FI	FN	MI	MN	MP	MC
RMD	377.6	13.5	7.9	13.6	15.5	16.8	34.9	10.1	18.1	53.1
IAM	**386.8**	13.9	**9.9**	16.2	18.4	19.0	**37.0**	12.7	24.7	58.4
DAT	371.7	8.5	**9.9**	14.9	**20.3**	**22.5**	34.0	12.3	**25.0**	**62.8**
GTS	365.9	13.7	7.7	15.6	18.1	20.0	33.9	13.5	19.5	60.1
GT	384.4	**17.3**	9.0	**16.3**	18.0	20.1	32.6	**14.5**	23.4	62.4

a) Impact of warm-up phase: As seen in Table III(a), the warm-up phase fixed some inaccurate mappings in RMD and IAM. Specifically, it mainly fixed wrong mappings of identical statements caused by RMD and missing token mappings in identical expressions caused by IAM. As seen in Table III(b), the warm-up phase took much longer than each fixing pass. While the fixing passes traverse the pruned mapped node sets, the warm-up phase requires a complete traversal of AST_1 and AST_2. It is due to the necessary initializations and optimizations during the warm-up phase that the fixing passes can be performed effectively and efficiently.

b) Impact of fixing passes: Table III(a) shows that the number of diffs that different fixing passes took effect on was widely distributed across different techniques, reflecting the differences in the types of inaccurate mappings in different techniques. $MatchUnique$ contributed significantly to RMD and GTS, mainly because they had difficulty matching statements with large changes and failed to effectively utilize the context of mapped composite statements, which $MatchUnique$ effectively handled. $MatchChildren$ contributed significantly to IAM and GT by fixing their missing token mappings. $MatchInner$ and $MatchParent$ contributed more to RMD and IAM, while $FixChildren$ and $FixNearby$ contributed more to DAT, GTS, and GT, which reflected the difference between techniques targeting accuracy and techniques targeting ES size: the former better avoided arbitrary mappings and wrong mappings, but was prone to missing mappings; the latter reduced missing mappings, but caused more arbitrary mappings and wrong mappings. $FixInner$ was effective in fixing arbitrary mappings of DAT, and $MatchNearby$ was effective in fixing missing mappings of GTS. $FixParent$ worked in some cases where the child nodes were mapped correctly and the parent nodes were not.

Table III(b) shows that $MatchChildren$, $MatchInner$, and $MatchParent$ took up the most time in most techniques. The time cost is mainly related to the number of analyzed nodes when identifying inaccurate mappings, and, secondly, to the number of inaccurate mappings that need to be fixed.

Answer to RQ4: Different fixing passes in DiffFix were able to fix inaccurate mappings, with contributions varying across techniques. Their execution time remained much shorter than the warm-up phase, which bore the main time cost for initialization and optimization.

G. Threat to Validity

Internal validity. Observations in Section III were derived based on the inaccurate diffs generated by five ASTDiff techniques on the DiffBenchmark dataset, which may not cover all characteristics of inaccurate mappings. However, the five techniques are the state-of-the-art ASTDiff techniques, which reflect current practice despite the tool-specific limitations. Moreover, DiffBenchmark was constructed upon Defects4J and RefOracle, which consists of commits from real-world popular and representative projects. Therefore, we are confident that the observations have covered most common inaccurate mappings.

External validity. The design and evaluation of DiffFix were mainly based on DiffBenchmark. To measure DiffFix ’s generalization capability, we applied it to fix the diffs generated by RMD, the most advanced ASTDiff technique, on 10,711 commits from 116 popular GitHub projects beyond DiffBenchmark. The manual verification of 320 randomly selected fixes achieved a precision of 93.12%, suggesting the high generalizability of DiffFix. Although this study was conducted on Java, the overall idea can be generalized to other languages with intuitive adaptations based on the language’s AST structure and type information.

VI. Discussion

Main implications. Firstly, identifying inaccurate mappings, as well as establishing accurate node mappings, requires the comprehensive utilization of multiple context and type information. Secondly, coexisting inaccurate mappings can be effectively and efficiently resolved in an incremental and iterative approach. Finally, inaccurate mappings exhibit certain frequent and generalizable patterns, suggesting that building a continuously updated dataset of inaccurate mappings would facilitate further research.

Impacts on downstream tasks. The accuracy of ASTDiff can have a significant impact on downstream tasks. However, there is still a lack of such evaluations. For example, in program repair scenarios [11], [42], accurate AST diffs tend to produce more reasonable code changes rather than arbitrary edit actions, which helps derive understandable repair patterns. Moreover, in machine learning scenarios [8], meaningful code changes offer clearer and more consistent features, as opposed to noise introduced by inaccurate diffs. These benefits warrant further investigation in future work.

VII. Conclusion

In this paper, we proposed an incremental AST diff fixing approach DiffFix based on the node’s context and type constraints. DiffFix iteratively identifies and fixes inaccurate mappings in a generated diff through nine deliberatively designed fixing passes. Experimental results on DiffBenchmark showed that DiffFix can significantly improve the node mapping accuracy of five state-of-the-art ASTDiff techniques with negligible time overhead.

For future work, we will introduce lightweight semantic analysis to DiffFix for identifying and fixing subtle inaccurate mappings involving method invocations. Also, it is worthwhile to investigate how to fix multi-mappings, which reflect some code cloning and merging activities beyond typical one-to-one mappings. Given the growing use of ASTDiff in code change analysis and automation research, it is imperative to conduct systematic studies on how ASTDiff accuracy affects downstream tasks and how DiffFix can benefit them.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under grants 62272037, 61872039, and 62502030, and Fundamental Research Funds for the Central Universities FRF-TP-25-030.

References

[1] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, “How do software engineers understand code changes? an exploratory study in industry,” in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11. [Online]. Available: https://doi.org/10.1145/2393596.2393656

[2] B. Fluri, M. Wursch, M. PInzger, and H. Gall, “Change distilling: Tree differencing for fine-grained source code change extraction,” *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007. [Online]. Available: https://doi.org/10.1109/TSE.2007.70731

[3] 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 (ASE)*, 2014, pp. 313–324. [Online]. Available: https://doi.org/10.1145/2642937.2642982

[4] J.-R. Falleri and M. Martinez, “Fine-grained, accurate and scalable source differencing,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3597503.3639148

[5] G. Dotzler and M. Philippsen, “Move-optimized source code tree differencing,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 660–671. [Online]. Available: https://doi.org/10.1145/2970276.2970315

[6] N. Zhang, Q. Chen, Z. Zheng, and Y. Zou, “iASTMapper: An iterative similarity-based abstract syntax tree mapping algorithm,” in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 863–874. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00178

[7] P. Alikhanifard and N. Tsantalis, “A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–63, 2025. [Online]. Available: https://doi.org/10.1145/3696002

[8] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, “On learning meaningful code changes via neural machine translation,” in *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019, pp. 25–36. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00021

[9] M. Janke and P. Mäder, "FS$^3_{\text{change}}$: A scalable method for change pattern mining," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3616–3629, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3269500

[10] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019. [Online]. Available: https://doi.org/10.1145/3360585

[11] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "FixMiner: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09780-z

[12] M. Wang, Z. Lin, Y. Zou, and B. Xie, "CoRA: Decomposing and describing tangled code changes for reviewer," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1050–1061. [Online]. Available: https://doi.org/10.1109/ASE.2019.00101

[13] X. Gao, A. Radhakrishna, G. Soares, R. Shariffdeen, S. Gulwani, and A. Roychoudhury, "APIfix: Output-oriented program synthesis for combating breaking changes in libraries," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021. [Online]. Available: https://doi.org/10.1145/3485538

[14] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating accurate and compact edit scripts using tree differencing," in *Proceedings of 34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 264–274. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00036

[15] Y. Fan, X. Xia, D. Lo, A. E. Hassan, Y. Wang, and S. Li, "A differential testing approach for evaluating abstract syntax tree mapping algorithms," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021, pp. 1174–1185. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00108

[16] M. Martinez, J.-R. Falleri, and M. Monperrus, "Hyperparameter optimization for AST differencing," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4814–4828, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3315935

[17] "Clang Static Analyzer," https://clang.llvm.org/docs/ClangStaticAnalyzer.html, 2025.

[18] "Eclipse JDT," https://github.com/eclipse-jdt/eclipse.jdt.core, 2025.

[19] "ESLint," https://eslint.org/, 2025.

[20] P. Bille, "A survey on tree edit distance and related problems," *Theoretical Computer Science*, vol. 337, no. 1, pp. 217–239, 2005. [Online]. Available: https://doi.org/10.1016/j.tcs.2004.12.030

[21] S. S. Chawathe and H. Garcia-Molina, "Meaningful change detection in structured data," in *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data (MOD)*, 1997, pp. 26–37. [Online]. Available: https://doi.org/10.1145/253260.253266

[22] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991. [Online]. Available: https://doi.org/10.1002/spe.4380210706

[23] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974. [Online]. Available: https://doi.org/10.1145/321796.321811

[24] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005. [Online]. Available: https://doi.org/10.1145/1082983.1083143

[25] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting similar java classes using tree algorithms," in *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR)*, 2006, pp. 65–71. [Online]. Available: https://doi.org/10.1145/1137983.1138000

[26] R. Cottrell, J. J. C. Chang, R. J. Walker, and J. Denzinger, "Determining detailed structural correspondence for generalization tasks," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, 2007, pp. 165–174. [Online]. Available: https://doi.org/10.1145/1287624.1287649

[27] R. Cottrell, R. J. Walker, and J. Denzinger, "Semi-automating small-scale source code reuse via structural correspondence," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2008, pp. 214–225. [Online]. Available: https://doi.org/10.1145/1453101.1453130

[28] M. Hashimoto and A. Mori, "Diff/TS: A tool for fine-grained structural change analysis," in *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE)*, 2008, pp. 279–288. [Online]. Available: https://doi.org/10.1109/WCRE.2008.44

[29] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008–1026, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.90

[30] A. Duley, C. Spandikow, and M. Kim, "Vdiff: A program differencing algorithm for verilog hardware description language," *Automated Software Engineering*, vol. 19, no. 4, pp. 459–490, 2012. [Online]. Available: https://doi.org/10.1007/s10515-012-0107-6

[31] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, "A retrospective of ChangeDistiller: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 51, no. 3, pp. 852–857, 2025. [Online]. Available: https://doi.org/10.1109/TSE.2025.3538326

[32] M. Chilowicz, E. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," in *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC)*, 2009, pp. 243–247. [Online]. Available: https://doi.org/10.1109/ICPC.2009.5090050

[33] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance," *ACM Transactions on Database Systems*, vol. 40, no. 1, pp. 1–40, 2015. [Online]. Available: https://doi.org/10.1145/2699485

[34] 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 (ASE)*, 2017, pp. 532–542. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115664

[35] 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 ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 679–690. [Online]. Available: https://doi.org/10.1145/3238147.3238219

[36] J. Matsumoto, Y. Higo, and S. Kusumoto, "Beyond GumTree: A hybrid approach to generate edit scripts," in *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2019, pp. 550–554. [Online]. Available: https://doi.org/10.1109/MSR.2019.00082

[37] C. Yang and E. J. Whitehead, "Pruning the AST with hunks to speed up tree differencing," in *Proceedings of 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 15–25. [Online]. Available: https://doi.org/10.1109/SANER.2019.8668032

[38] A. Fujimoto, Y. Higo, J. Matsumoto, and S. Kusumoto, "Staged tree matching for detecting code move across files," in *Proceedings of the 28th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2020, pp. 396–400. [Online]. Available: https://doi.org/10.1145/3387904.3389289

[39] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "HyperDiff: Computing source code diffs at scale," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 288–299. [Online]. Available: https://doi.org/10.1145/3611643.3616312

[40] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[41] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3007722

[42] Y. Tang, H. Chen, Z. He, and H. Zhong, "Understanding mirror bugs in multiple-language projects," *ACM Transactions on Software Engineering and Methodology*, pp. 1–26, 2025, just accepted. [Online]. Available: https://doi.org/10.1145/3727145