A Differential Testing Approach for Evaluating Abstract Syntax Tree Mapping Algorithms

Yuanrui Fan[*][†], Xin Xia[‡][§], David Lo[¶], Ahmed E. Hassan[||], Yuan Wang[**], Shanping Li[*]
[*]Zhejiang University, China; [†]PengCheng Laboratory, China; [‡]Monash University, Australia;
[¶]Singapore Management University, Singapore; [||]Queen's University, Canada; [**]Huawei Sweden Research Center
{yrfan, shan}@zju.edu.cn, Xin.Xia@monash.edu, davidlo@smu.edu.sg, ahmed@cs.queensu.ca, Yuan.Wang1@huawei.com

***Abstract*—Abstract syntax tree (AST) mapping algorithms are widely used to analyze changes in source code. Despite the foundational role of AST mapping algorithms, little effort has been made to evaluate the accuracy of AST mapping algorithms, i.e., the extent to which an algorithm captures the evolution of code. We observe that a program element often has only one best-mapped program element. Based on this observation, we propose a hierarchical approach to automatically compare the similarity of mapped statements and tokens by different algorithms. By performing the comparison, we determine if each of the compared algorithms generates inaccurate mappings for a statement or its tokens. We invite 12 external experts to determine if three commonly used AST mapping algorithms generate accurate mappings for a statement and its tokens for 200 statements. Based on the experts' feedback, we observe that our approach achieves a precision of 0.98–1.00 and a recall of 0.65–0.75. Furthermore, we conduct a large-scale study with a dataset of ten Java projects containing a total of 263,165 file revisions. Our approach determines that GumTree, MTDiff and IJM generate inaccurate mappings for 20%–29%, 25%–36% and 21%–30% of the file revisions, respectively. Our experimental results show that state-of-the-art AST mapping algorithms still need improvements.**

***Index Terms*—Program element mapping, abstract syntax trees, software evolution**

I. Introduction

Program element mapping algorithms are the underlying basis for analyzing changes between two versions of a source code file (i.e., a file revision) [20]. Abstract syntax tree (AST) mapping algorithms represent a file revision and program elements of the file as two abstract syntax trees (ASTs) and nodes, respectively [8], [9], [12]. The algorithms approximate the similarity of nodes and calculate mappings of nodes between the two ASTs. We define accurate mappings as mappings that can reflect the evolution of code well.

Edit actions including adding, deleting, moving and updating nodes of an AST can be calculated based on the generated mappings by an AST mapping algorithm [6]. Such edit actions can describe changes to the syntactic structure of code, e.g., parameters added in a method call can be represented as added nodes in an AST. Accurate mappings lead to accurate edit actions that can reflect a developer's intent. Many prior studies apply AST mapping algorithms to calculate edit actions for further analyses, e.g., API recommendation [27], mining code change patterns [22], and automated program repair [32]. The accuracy of the used AST mapping algorithms is vital for the correctness of the proposed approaches by prior studies.

However, evaluations of the accuracy of AST mapping algorithms are limited. When evaluating the generated mappings by an algorithm, prior work relies heavily on manual analysis of the derived edit actions from the mappings [8], [9], [12]. Since it is infeasible to analyze all file revisions by hand, prior studies select a small sample for analysis. The many cases where AST mapping algorithms perform badly cannot be revealed. Furthermore, manually analyzing the mappings of program elements is time-consuming and tedious. An approach that can automatically find the inaccurate mappings as generated by AST mapping algorithms would be helpful. Practitioners and researchers can leverage such an approach to explore and navigate the generated mappings by an algorithm before performing further analyses.

In this paper, we propose an approach for evaluating AST mapping algorithms. We observe that a program element e often has only one best-mapped program element $\hat{e}$ in a file revision. The element $\hat{e}$ might be empty, i.e., e should not be mapped. If two algorithms inconsistently map such an element, at least one of the algorithms inaccurately mapped the element. We define *similarity* of two program elements in a file revision as the likelihood of the two elements to be mapped. Our idea is that if an algorithm maps a more similar program element for e than another algorithm, the latter algorithm is likely to have made a mistake. We aim to automatically compare the similarity of mapped program elements by different AST mapping algorithms. In the software testing area, this approach is referred to as *differential testing* [25].

To ease analysis, we refine the mappings of AST nodes into mappings of each statement and tokens in the statement. And we treat each statement as an analysis unit. By doing so, we avoid analyzing the mappings of AST nodes at all granularity levels. Notice that statements include declarations (e.g., method declarations) in our paper. A token is defined as a sequence of characters representing a program element such that none of its subsequences represents a program element. Given two algorithms, if they inconsistently map a statement or its tokens, we refer to such statements as *statements with inconsistent mappings* for the two algorithms. If an algorithm inaccurately maps a statement or its tokens, we refer to such statements as *statements with inaccurate mappings* for the

[§]Corresponding author.

algorithm.

We analyzed three AST mapping algorithms in our paper, namely GumTree [9], MTDiff [8] and IJM (Iterative Java Matcher) [12]. As shorthand notations, we use GT and MTD to represent GumTree and MTDiff, respectively. We manually analyze 575 statements with inconsistent mappings for comparing the algorithms. Through the manual analysis, we design a hierarchical approach to automatically compare the similarity of mapped statements and tokens by different algorithms. This hierarchical approach uses six measures collectively to perform the comparison. By performing the comparison, we determine statements with inaccurate mappings for each of the compared algorithms.

We invite 12 external experts to determine if the studied algorithms generate inaccurate mappings (at the statement or token level) for 200 statements. Compared to the experts' evaluation, we find that our approach achieves a precision of 0.98–1.00 and a recall of 0.65–0.75. We run the studied algorithms on all the file revisions of ten Java projects. We use our approach to analyze the generated mappings by the algorithms. For 20%–29%, 25%–36% and 21%–30% of the file revisions, GT, MTD and IJM are determined to generate inaccurate mappings, respectively. The results show that state-of-the-art AST mapping algorithms still need improvements. We make our code and data publicly available on [1].

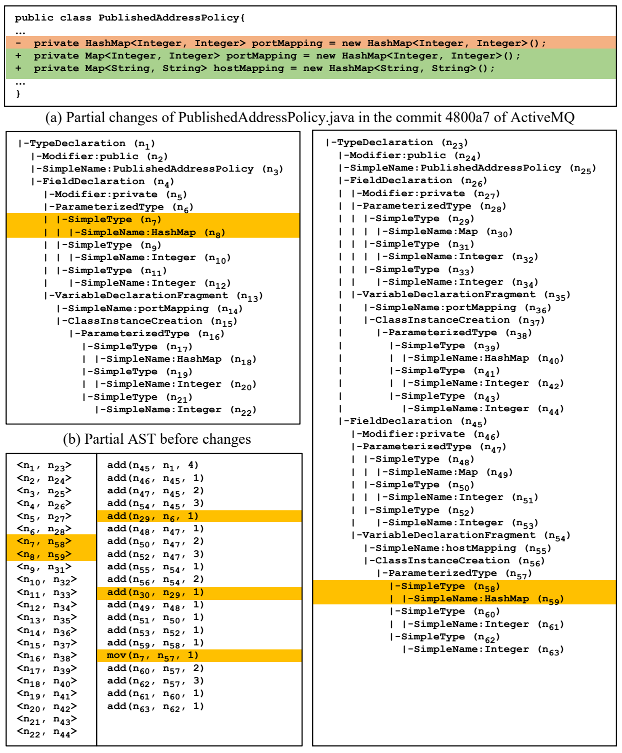
Our contributions are summarized as follows:

- We propose an approach that can automatically detect statements with inaccurate mappings for AST mapping algorithms. Almost all of the statements with inaccurate mappings as determined by our approach are also determined as such by experts (98%–100%).
- We use our approach to analyze the generated mappings by GT, MTD and IJM for 263,165 file revisions. The three algorithms are determined to generate inaccurate mappings for a considerable number of file revisions.

II. Preliminaries

Abstract syntax tree. A source code file can be parsed as an abstract syntax tree (AST). An AST is a labeled ordered rooted tree, which is composed of a set of nodes that are connected by edges. An edge represents a parent-child relationship. A node n_1 is the *parent* of another node n_2, if n_2 is a child of n_1. The node that has no parent is called the *root node*. A node that has no child is called a *leaf node*. For a node, the nodes along the path to the root node are called its *ancestors*. And the node is called their *descendant*. Each node in an AST represents a code element (e.g., a statement) with a *label* to indicate its type. Some nodes have a *value* to indicate the corresponding tokens of the element.

Example 2.1. Fig. 1(b) and Fig. 1 present the two ASTs before and after the code changes at lines 1–3 in Fig. 1(a). The ASTs are built using GT [9]. We only show partial ASTs for clarity. The AST in Fig. 1(b) contains 22 nodes. The node n_4 has three child nodes n_5, n_6 and n_{13}, and its label is `FieldDeclaration`. The label of n_5 is `Modifier`, and



```
  public class PublishedAddressPolicy{
  …
1 -  private HashMap<Integer, Integer> portMapping = new HashMap<Integer, Integer>();
2 +  private Map<Integer, Integer> portMapping = new HashMap<Integer, Integer>();
3 +  private Map<String, String> hostMapping = new HashMap<String, String>();
  …
  }
```

(a) Partial changes of PublishedAddressPolicy.java in the commit 4800a7 of ActiveMQ

```
|-TypeDeclaration (n1)
  |-Modifier:public (n2)
  |-SimpleName:PublishedAddressPolicy (n3)
  |-FieldDeclaration (n4)
    |-Modifier:private (n5)
    |-ParameterizedType (n6)
    | |-SimpleType (n7)
    | | |-SimpleName:HashMap (n8)
    | |-SimpleType (n9)
    | | |-SimpleName:Integer (n10)
    | |-SimpleType (n11)
    |   |-SimpleName:Integer (n12)
    |-VariableDeclarationFragment (n13)
      |-SimpleName:portMapping (n14)
      |-ClassInstanceCreation (n15)
        |-ParameterizedType (n16)
          |-SimpleType (n17)
          | |-SimpleName:HashMap (n18)
          |-SimpleType (n19)
          | |-SimpleName:Integer (n20)
          |-SimpleType (n21)
            |-SimpleName:Integer (n22)
```

(b) Partial AST before changes

```
<n1, n23>     add(n45, n1, 4)
<n2, n24>     add(n46, n45, 1)
<n3, n25>     add(n47, n45, 2)
<n4, n26>     add(n54, n45, 3)
<n5, n27>     add(n29, n6, 1)
<n6, n28>     add(n48, n47, 1)
<n7, n58>     add(n50, n47, 2)
<n8, n59>     add(n52, n47, 3)
<n9, n31>     add(n55, n54, 1)
<n10, n32>    add(n56, n54, 2)
<n11, n33>    add(n30, n29, 1)
<n12, n34>    add(n49, n48, 1)
<n13, n35>    add(n51, n50, 1)
<n14, n36>    add(n53, n52, 1)
<n15, n37>    add(n59, n58, 1)
<n16, n38>    mov(n7, n57, 1)
<n17, n39>    add(n60, n57, 2)
<n18, n40>    add(n62, n57, 3)
<n19, n41>    add(n61, n60, 1)
<n20, n42>    add(n63, n62, 1)
<n21, n43>
<n22, n44>
```

(d) Mappings and edit actions by GT

```
|-TypeDeclaration (n23)
  |-Modifier:public (n24)
  |-SimpleName:PublishedAddressPolicy (n25)
  |-FieldDeclaration (n26)
  | |-Modifier:private (n27)
  | |-ParameterizedType (n28)
  | | |-SimpleType (n29)
  | | | |-SimpleName:Map (n30)
  | | |-SimpleType (n31)
  | | | |-SimpleName:Integer (n32)
  | | |-SimpleType (n33)
  | |   |-SimpleName:Integer (n34)
  | |-VariableDeclarationFragment (n35)
  |   |-SimpleName:portMapping (n36)
  |   |-ClassInstanceCreation (n37)
  |     |-ParameterizedType (n38)
  |       |-SimpleType (n39)
  |       | |-SimpleName:HashMap (n40)
  |       |-SimpleType (n41)
  |       | |-SimpleName:Integer (n42)
  |       |-SimpleType (n43)
  |         |-SimpleName:Integer (n44)
  |-FieldDeclaration (n45)
    |-Modifier:private (n46)
    |-ParameterizedType (n47)
    | |-SimpleType (n48)
    | | |-SimpleName:Map (n49)
    | |-SimpleType (n50)
    | | |-SimpleName:Integer (n51)
    | |-SimpleType (n52)
    |   |-SimpleName:Integer (n53)
    |-VariableDeclarationFragment (n54)
      |-SimpleName:hostMapping (n55)
      |-ClassInstanceCreation (n56)
        |-ParameterizedType (n57)
          |-SimpleType (n58)
          | |-SimpleName:HashMap (n59)
          |-SimpleType (n60)
          | |-SimpleName:Integer (n61)
          |-SimpleType (n62)
            |-SimpleName:Integer (n63)
```

(c) Partial AST after changes

Fig. 1: Mappings of AST nodes and edit actions as calculated by GT for the partial changes of a file in the commit 4800a7 of ActiveMQ.

it has a string value `private` indicating the field declaration is private for the class.

AST mapping algorithms. Given a file revision, an AST mapping algorithm parses the file before and after the revision as two ASTs. Let us denote the two ASTs as *source AST* and *target AST*, respectively. By approximating the similarity of nodes, the algorithm calculates mappings of nodes between the two ASTs. Only the nodes that have the same label can be mapped. The mappings are a set of pairs $\langle n_s, n_t \rangle$, in which n_s belongs to the source AST and n_t belongs to the target AST.

Example 2.2. We leverage GT to calculate mappings of nodes between the ASTs shown in Fig. 1(b) and Fig. 1(c). Fig. 1(d) presents the mappings in the first column. Specially, n_4 is mapped to n_{26}, indicating that the field declaration `portMapping` at line 1 in Fig. 1(a) is mapped to the field declaration `portMapping` at line 2 in Fig. 1(a). In addition, n_7 and n_8 are mapped to n_{58} and n_{59}, respectively. Such mappings indicate that the first `HashMap` of line 1 and the `HashMap` of line 3 in Fig. 1(a) are mapped, i.e., GT considers that the two `HashMap` tokens are the same element across the file revision.

Edit actions. Based on mappings of AST nodes, a series of edit actions can be calculated to transform the source AST to the target AST [6]. Prior studies apply the Chawathe et al.' algorithm [6] to calculate the edit actions [8], [9], [12].

Generally, there are four types of edit actions:

- $upd(n, v)$ replaces the value of the node n with a value v.
- $add(n, p, i)$ adds a node n as the i^{th} child of the node p if p is not null. Otherwise, the node n is added as the new root node.
- $del(n)$ deletes a leaf node n.
- $mov(n, p, i)$ moves the node n as the i^{th} child of the node p. The subtree rooted at n is moved together with n.

Example 2.3. Fig. 1(d) presents the edit actions that are calculated based on the mappings shown in the same figure. We use GT's implementation of Chawathe et al.'s algorithm. GT produces a sequence of 20 edit actions. One of the 20 actions is $mov(n_7, n_{57}, 1)$. This action moves the first `HashMap` of line 1 in Fig. 1(a) to line 3 in Fig. 1(a).

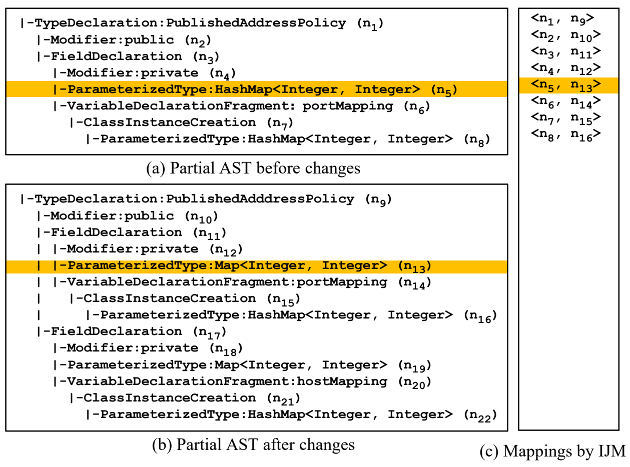
Current evaluations of AST mapping algorithms. Current evaluations of AST mapping algorithms include automatic and manual evaluations [8], [9], [12]. They are both based on the edit actions that are calculated from the generated mappings by different algorithms.

The number of edit actions is commonly used as an automatic measure for estimating the cognitive load for a developer when understanding the essence of a file revision [8], [9], [12]. Mappings with fewer edit actions are considered to be better. However, the number of edit actions cannot reflect the accuracy of the mappings [12]. Prior work relies heavily on manual analyses of the edit actions to determine the accuracy of the mappings [8], [9], [12]. Frick et al. [12] proposed three criteria for determining if a mapping is accurate: (1) each mapping should be comprehensible, i.e., why the two program elements should be mapped; (2) the generated edit actions should be helpful in understanding the changes; (3) there exists no other comprehensible mappings resulting in fewer actions.

Example 2.4. In Fig. 1(d), we determine the accuracy of the mappings and highlight the inaccurate mappings with a yellow background. The mappings $\langle n_7, n_{58}\rangle$ and $\langle n_8, n_{59}\rangle$ are determined to be inaccurate. The nodes in the ASTs and edit actions that relate to the inaccurate mappings are also highlighted. We consider that it is not comprehensible to map the first `HashMap` of line 1 to the `HashMap` of line 3 in Fig. 1(a). Also, the action $mov(n_7, n_{57}, 1)$ is not helpful in understanding the changes. It is better to map n_7 and n_8 to n_{29} and n_{30}, i.e., map the first `HashMap` of line 1 and the `Map` of line 2 in Fig. 1(a). Thus, the mappings for the nodes n_{29} and n_{30} are also considered to be inaccurate.

III. Motivation

Prior evaluations of AST mapping algorithms are limited to manually analyzing a small sample of file revisions [8], [9], [12]. For instance, to evaluate GT, Falleri et al. manually analyzed 144 file revisions [9]. To evaluate MTD, Dotzler et al. manually analyzed 10 file revisions [8]. The many cases where an AST mapping algorithm performs badly cannot be revealed. An approach that can automatically find inaccurate



```
|-TypeDeclaration:PublishedAddressPolicy (n1)
  |-Modifier:public (n2)
  |-FieldDeclaration (n3)
    |-Modifier:private (n4)
    |-ParameterizedType:HashMap<Integer, Integer> (n5)
    |-VariableDeclarationFragment: portMapping (n6)
      |-ClassInstanceCreation (n7)
        |-ParameterizedType:HashMap<Integer, Integer> (n8)
```

(a) Partial AST before changes

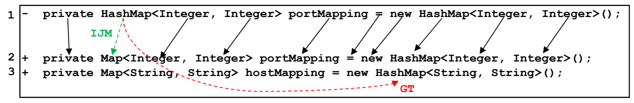
```
|-TypeDeclaration:PublishedAdddressPolicy (n9)
  |-Modifier:public (n10)
  |-FieldDeclaration (n11)
  | |-Modifier:private (n12)
  | |-ParameterizedType:Map<Integer, Integer> (n13)
  | |-VariableDeclarationFragment:portMapping (n14)
  |   |-ClassInstanceCreation (n15)
  |     |-ParameterizedType:HashMap<Integer, Integer> (n16)
  |-FieldDeclaration (n17)
    |-Modifier:private (n18)
    |-ParameterizedType:Map<Integer, Integer> (n19)
    |-VariableDeclarationFragment:hostMapping (n20)
      |-ClassInstanceCreation (n21)
        |-ParameterizedType:HashMap<Integer, Integer> (n22)
```

(b) Partial AST after changes

```
<n1, n9>
<n2, n10>
<n3, n11>
<n4, n12>
<n5, n13>
<n6, n14>
<n7, n15>
<n8, n16>
```

(c) Mappings by IJM

Fig. 2: Mappings of AST nodes calculated by IJM for the changes shown in Fig. 1(a).



```
1 -  private HashMap<Integer, Integer> portMapping = new HashMap<Integer, Integer>();
2 +  private Map<Integer, Integer> portMapping = new HashMap<Integer, Integer>();
3 +  private Map<String, String> hostMapping = new HashMap<String, String>();
```

Fig. 3: The generated mappings by GT and IJM in Fig. 1 and Fig. 2. Dashed lines show the inconsistent mappings.

mappings as generated by an algorithm would be helpful for both researchers and practitioners.

We observe that for a file revision, a program element e_1 often has only one best-mapped element e_2. The element e_2 can be empty, indicating that e_1 should not be mapped. For instance, in Fig. 1(a), the first `HashMap` of line 1 should be mapped to the `Map` of line 2. If two algorithms inconsistently map such a program element, at least one of the algorithms inaccurately maps the element. The inaccurate mappings can be found by analyzing the inconsistencies between the two algorithms.

We provide an example. For the file revision shown in Fig. 1(a), we use IJM to calculate mappings of AST nodes. Different AST mapping algorithms may use different ASTs to represent the same file. Fig. 2(a) and (b) show the IJM's ASTs that represent the changes at lines 1-3 in Fig. 1(a). Fig. 2(c) presents the generated mappings by IJM. The value of n_5 denotes the first `HashMap<Integer, Integer>` of line 1 in Fig. 1(a). The value of n_{13} denotes the `Map<Integer, Integer>` of line 2 in Fig. 1(a). As shown in Fig. 2(c), IJM maps n_5 to n_{13}. By further mapping the tokens in the value of the two nodes, we find that IJM accurately maps the first `HashMap` of line 1 in Fig. 1(a). This mapping is inconsistent with the generated mappings by GT. Fig. 3 visualizes the generated mappings by GT and IJM and their inconsistent mappings.

We find that both GT and IJM map the statement at line 1 to the statement at line 2 in Fig. 3. The first `HashMap` of line 1 and the `Map` of line 2 belong to the mapped statements.

They both denote the type of the field `portMapping`. On the other hand, the first `HashMap` of line 1 and the `HashMap` of line 3 belong to unmapped statements. Thus, in comparison to the `HashMap` of line 3, the `Map` of line 2 is more similar to the first `HashMap` of line 1. Finally, the inaccurately mapped `HashMap` by GT is detected. In this motivational example, we attempt to *automatically* perform the above analysis and find the inaccurate mappings as generated by AST mapping algorithms.

IV. Studied AST Mapping Algorithms

In this study, we analyze three state-of-the-art AST mapping algorithms, namely GumTree [9], MTDiff [8] and IJM [12]. We briefly introduce the three algorithms below.

GumTree (GT) is proposed by Falleri et al [9]. Given two ASTs, GT matches nodes between the ASTs in two phases: In the first phase, GT applies a greedy top-down algorithm to search and map identical subtrees. In the second phase, GT applies a bottom-up algorithm to map a pair of nodes between the two ASTs if they share a significant number of mapped descendants. Then, GT tries to map previously unmapped descendants of those nodes.

MTDiff (MTD) is proposed by Dotzler et al. [8]. MTD is based on the ChangeDistiller algorithm [11]. First, MTD applies the identical subtree optimization to reduce the mapping problem by removing unchanged subtrees from the ASTs. Then, MTD maps nodes using the ChangeDistiller algorithm. Another four optimizations are finally applied to find additional mappings of nodes.

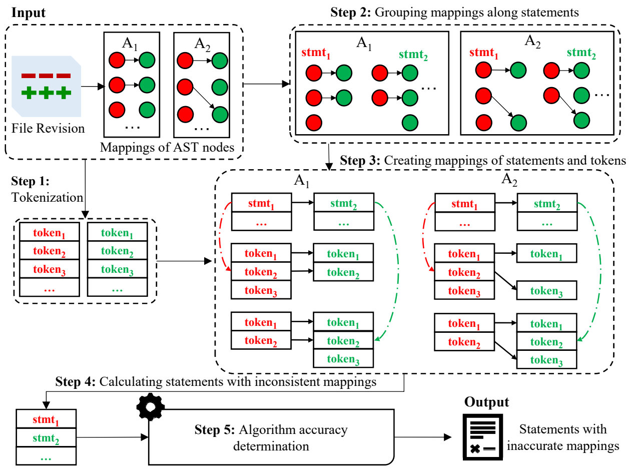
IJM is proposed by Frick et al. [12]. IJM is an AST mapping algorithm specialized for Java. In comparison to GT, IJM works on a reduced AST, in which many name nodes are pruned and the value of each pruned node is merged to its parent node. Then, IJM splits the AST into parts along each declaration. Finally, it maps AST nodes from the corresponding parts between the two ASTs using an adaptation of the GT algorithm. The adaptation adds name-awareness to GT, which considers similarity of values when mapping two nodes.

In this study, we apply the implementations of the three algorithms that are provided on GitHub [2]–[4].

V. Approach

In this section, we describe the details of our approach. Notice that our approach is general. We implement it for Java programs but we can modify it to support other programs as long as we can generate ASTs from them.

The granularity of AST nodes ranges from a statement to a single literal. It is too complex to analyze mappings of AST nodes of all granularity levels. Furthermore, finding the corresponding nodes from the used ASTs by different algorithms is a big challenge. Because different algorithms may use ASTs with different sets of nodes (including leaf nodes) to represent the same file, as shown in Fig. 1 and Fig. 2. We observe that different algorithms commonly encode statements as AST nodes. Additionally, tokens of a file are



Input
File Revision
A_1
A_2
Mappings of AST nodes
Step 2: Grouping mappings along statements
$stmt_1$
$stmt_2$
Step 3: Creating mappings of statements and tokens
Step 1: Tokenization
$token_1$
$token_2$
$token_3$
Step 4: Calculating statements with inconsistent mappings
Step 5: Algorithm accuracy determination
Output
Statements with inaccurate mappings

Fig. 4: Overview of our approach. The red color and green color denote that the program element is from the file before and after the revision, respectively.

not impacted by the composing nodes of an AST. To solve the above issues, we refine the mappings of AST nodes into mappings of statements and tokens. Moreover, we treat each statement (including the statement itself and its tokens) as an analysis unit.

Fig. 4 presents the overview of our approach. Given the generated mappings by two algorithms (A_1 and A_2) for a file revision, we take five steps to calculate the statements with inaccurate mappings for each algorithm. In Sections V-A to V-E, we describe the five steps, respectively. In Section V-F, we elaborate how to compare the similarity of mapped statements and tokens by different algorithms.

A. Tokenization

In this step, we tokenize the file before and after a revision. Instead of using punctuation and spaces, we use the parsed AST of the source file to tokenize the file. For example, a string literal containing punctuation and spaces is considered as a single token.

For a Java file, we first use the Eclipse JDT parser to generate a standard JDT AST. Then, we extract the tokens from value of each AST node. Notice that we ignore AST nodes representing comments and Javadocs. Because comments and Javadocs are typically not treated as code [17]. As a result, we retrieve two token lists for a file before and after a revision, respectively. The token lists are not impacted by the used AST by each algorithm

B. Grouping Mappings along Statements

In this step, we separately group mappings of AST nodes along statements for A_1 and A_2. In an AST, a statement (e.g., a method declaration) can have descendant statements. For each statement in the source and target ASTs that are being analyzed by an algorithm, we first group the nodes that belong to the statement but do not belong to its descendant statements. For instance, our framework groups n_1, n_2, and n_3 in Fig. 1(b) for the type declaration `PublishedAddressPolicy`. Then, we find the generated mappings by the algorithm for the grouped nodes. These mappings are grouped for the statement.

C. Creating Mappings of Statements and Tokens

In this step, we separately calculate mappings of statements and tokens for A_1 and A_2. Among the grouped mappings for each statement, we can directly find the mapping for the statement. For instance, in Fig. 1(b), n_1 represents the type declaration `PublishedAddressPolicy` and the mapping of n_1 is considered as the mapping of the declaration.

For a token, we refer to an AST node whose corresponding program element contains the token as a *relevant node* of the token. Among the relevant nodes of a token, we refer to the node of the lowest level as the *directly relevant node* of the token. In return, we refer to the token as a *directly relevant token* of the node. We take the first `HashMap` of line 1 in Fig. 1(a) as an example. In the used AST by GT shown in Fig. 1(b), n_1, n_4, n_6, n_7 and n_8 are relevant nodes of the token. And the directly relevant node for the token is n_8. In the used AST by IJM shown in Fig. 2(a), n_1, n_3 and n_5 are relevant nodes of the token. And the directly relevant node for the token is n_5.

An AST node can have several directly relevant tokens. For instance, the node n_5 in Fig. 2(a) has three directly relevant tokens including a `HashMap` and two `Integer` tokens. Such tokens compose the value of n_5. Another example is n_1 in Fig. 2, which has two directly relevant tokens including the `class` and `PublishedAddressPolicy` tokens in Fig. 1(a). The `PublishedAddressPolicy` token is the value of the node, while the `class` token does not belong to its value.

We observe that a token's mapping is determined by the mapping of its directly relevant node. We also take the first `HashMap` of line 1 in Fig. 1(a) as an example. In Fig. 1, GT maps n_8 to n_{59}—indicating that the token is mapped to the `HashMap` of line 3 in Fig. 1(a). And in Fig. 2, IJM maps n_5 to n_{13}—indicating that the token can only be mapped to a directly relevant token of n_{13}.

For each node of the source and target ASTs, we calculate all the directly relevant tokens and list the tokens according to their character positions. Then, for each pair of mapped nodes, we map tokens from the lists of directly relevant tokens of the nodes. If both lists have only one token, we directly map two tokens. Otherwise, we separately map the tokens composing the value of the nodes and other tokens, since tokens composing the value of a node can only be mapped to those composing the value of another node. For the tokens composing the values of the two nodes, we first sequentially map identical tokens between the two lists and then map the tokens (including the first and last tokens) that are surrounded by already mapped pairs of tokens. Mapping program elements surrounded by already mapped pairs is a commonly used heuristic by program element mapping algorithms [20]. After that, our framework applies the same method to map the tokens that do not belong to the value of the two nodes.

For instance, in Fig. 2, n_5 and n_{13} are mapped. The directly relevant tokens of n_5 include `HashMap`, `Integer` and `Integer`. And the directly relevant tokens of n_{13} include `Map`, `Integer` and `Integer`. All of the tokens belong to the values of the two nodes. We first sequentially map the two `Integer` tokens between the two lists. `HashMap` and `Map` are surrounded by already mapped tokens and they are further mapped.

As a result, we calculate mappings of tokens using the generated mappings of the AST nodes by A_1 and A_2. We further group mappings of tokens along each statement based on the grouped mappings of AST nodes for the statement.

D. Calculating Statements with Inconsistent Mappings

In this step, we calculate statements with inconsistent mappings for comparing A_1 and A_2. For each statement in the file before and after the revision, we first calculate whether the two algorithms inconsistently map the statement. Then, for each token in the statement, we calculate whether the two algorithms inconsistently map the token. Finally, we output statements with inconsistent mappings and for each statement, we group the inconsistent mappings of the statement and its tokens by comparing the two algorithms.

E. Algorithm Accuracy Determination

In this step, we compare the similarity of mapped statements and tokens across each pair of different algorithms. By performing the comparison, we determine the accuracy of each algorithm in the mapping of a statement or a token. In this section, we introduce how to determine the accuracy of each algorithm by performing the comparison. In Section V-F, we elaborate how to compare the similarity of mapped statements and tokens by different algorithms.

Let us denote the similarity between two program elements e and $\hat{e}$ as $Sim(e, \hat{e})$. The elements can be statements or tokens. Suppose that two algorithms (A_0 and A_1) map a program element e_0 to two different elements e_1 and e_2, respectively. We notice that the two algorithms also inconsistently map e_1 and e_2. Suppose that A_0 maps an element e_3 to e_2, and A_1 maps an element e_4 to e_1. In other words, A_0 generates two mappings $\{\langle e_0, e_1\rangle, \langle e_3, e_2\rangle\}$. And A_1 generates two mappings $\{\langle e_0, e_2\rangle, \langle e_4, e_1\rangle\}$.

Suppose that $Sim(e_0, e_1)$ is larger than $Sim(e_0, e_2)$, we determine that A_0 is more accurate than A_1 in mapping e_0. However, it is not enough to determine that the mapping of e_0 to e_1 is better than the generated mappings by A_1. We must also check the condition: $Sim(e_0, e_1) > Sim(e_4, e_1)$, i.e., A_0 is also more accurate than A_1 in mapping e_1. If this condition is also satisfied, we determine that A_1 inaccurately maps e_0 and e_1. Similarly, we can also determine if the two generated mappings by A_0 are inaccurate.

Given a pair of algorithms and a file revision, we calculate the statements with inconsistent mappings. For each statement with inconsistent mappings, we perform the above comparison

TABLE I: Statistics of the studied projects.

Projects	#Commit	#File Revision
ActiveMQ	8,059	24,813
Commons IO	1,067	2,727
Commons Lang	3,031	6,917
Commons Math	4,257	18,133
Junit4	1,241	3,802
Hibernate ORM	10,170	51,711
Hibernate Search	5,369	27,002
Spring Framework	14,754	68,413
Spring Roo	4,274	19,894
Netty	11,135	39,753
Total	63,357	263,165

for the mappings of the statement and its tokens as generated by the two algorithms that are being compared. Consequently, we calculate statements with inaccurate mappings for each algorithm.

For each studied algorithm, we separately compare it with the other two studied algorithms. Finally, we calculate a union set of statements with inaccurate mappings for any of the algorithms that are being compared.

F. Similarity Comparison

Our aim is to automatically compare the similarity of mapped statements and tokens by different algorithms. To realize this aim, we perform a manual analysis of statements with inconsistent mappings. Also, we verify if statements with inconsistent mappings can expose the inaccurate mappings as generated by the algorithms.

In this study, we analyze ten open-source Java projects. Table I presents statistics of the ten studied projects. These projects were analyzed by prior studies [12], [17]. We collect the commits of the projects from the creation date of the projects to January 2019.

We can compare three pairs of algorithms, namely GT vs. MTD, GT vs. IJM and MTD vs. IJM. For each pair of algorithms, we sample 50 file revisions for which the two algorithms inconsistently map program elements from the studied projects. For each file revision, we analyze all the statements before and after the revision that involve the inconsistent mappings. We analyze 178, 191 and 206 statements for comparing the three pairs of algorithms, respectively. In total, we analyze 575 statements. For each statement, we determine the accuracy of mappings as generated by each of compared algorithms using Frick et al.'s criteria [12]. In total, we make 1,150 (575×2) determinations.

The first and second author of this paper separately analyzed the statements with inconsistent mappings. Both authors have at least three years of programming experience in Java. We calculate Fleiss' Kappa [10] to estimate the agreement of the two annotators' determination results. The Kappa value is 0.81, which indicates an excellent agreement. Finally, the two annotators compared their determination results to uncover disagreements. For each statement with a disagreement, the annotators further discussed the accuracy of the generated mappings by the two compared algorithms.

We analyze all the statements with inaccurate mappings to design similarity measures to distinguish accurate and inaccurate mappings of statements and tokens. We categorize the statements with inaccurate mappings for the algorithms along each similarity measure. For each statement with inaccurate mappings, we consider if we have designed a measure that can identify the inaccurate mappings. If we have designed such a measure, we categorize the statement to the category of the measure. Otherwise, we try to design a new similarity measure to distinguish the accurate and inaccurate mappings. By doing so, we design hierarchical similarity measures for two statements and two tokens.

From our manual analysis, we have the following findings.

- Among the 178 statements for comparing GT and MTD, 71 and 129 statements involve inaccurate mappings as generated by GT and MTD, respectively. Among the 191 statements for comparing GT and IJM, 114 and 91 statements involve inaccurate mappings as generated by GT and IJM, respectively. Among the 206 statements for comparing MTD and IJM, 113 and 117 statements involve inaccurate mappings as generated by MTD and IJM, respectively.
- For each statement with inconsistent mappings for comparing two algorithms, at least one algorithm is determined to generate inaccurate mappings. Hence, statements with inconsistent mappings can expose the inaccurate mappings as generated by the algorithms.
- There exist cases where two algorithms consistently produce an inaccurate mapping of a statement or a token.

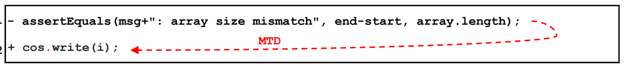
In total, we find 185 (71 + 114), 242 (129 + 113), 208 (91 + 117) statements with inaccurate mappings for GT, MTD and IJM, respectively. Based on our observation, we design two similarity measures for comparing two statements and four similarity measures for comparing two tokens. Table II presents the six measures. Notice that we define two tokens with the same value as *identical tokens*. We use the six similarity measures collectively to compare the generated mappings by different algorithms. In the remaining part of this section, we first introduce the six measures and our categorization results for the statements with inaccurate mappings. Then we describe the usage of the proposed measures.

The six similarity measures are elaborated below.

NIT is defined as the number of mapped identical tokens between a pair of mapped statements. Two mapped statements with a larger NIT are more similar and an NIT of 0 indicates that the two statements are highly dissimilar. If two statements are mapped with an NIT of 0, we determine the mapping as inaccurate. Otherwise, for a pair of mapped statements, we check if the mapping of one of the statements to another statement can achieve a larger NIT. For instance, in Fig. 5(a), MTD inaccurately maps the statements at lines 1 and 2, and the NIT is 0. In Fig. 5(b), GT maps the statements at lines 1 and 2 with an NIT of five. IJM maps the statements at lines 1 and 3 with an NIT of four. We determine that GT is more accurate than IJM in mapping the statement at line 1.

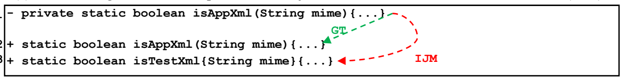
TABLE II: The similarity measures for statements and tokens.

Element	Measure	Measure Description	GT	MTD	IJM
Stmt.	NIT	Number of mapped identical tokens between a pair of mapped statements.	22	72	17
	PM	Whether the parent nodes of a pair of mapped statements are mapped.	9	18	7
Token	TYPE	Whether mapped tokens have the same type.	29	24	0
	STMT	Whether mapped tokens belong to a pair of mapped statements.	71	84	146
	VAL	Whether mapped tokens have the same value.	16	0	4
	LLCS	Length of the longest common subsequence calculated using mapped tokens between mapped statements.	10	10	4
Other			28	34	30



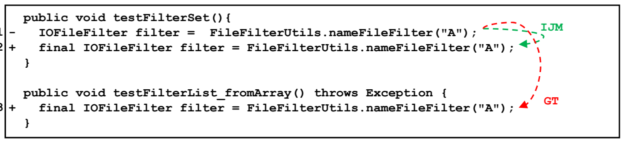
```
1 - assertEquals(msg+": array size mismatch", end-start, array.length);
2 + cos.write(i);
```

(a) Partial changes of CountOutputStreamTest.java from commit 4ab6a3 of Commons IO **(NIT)**



```
1 - private static boolean isAppXml(String mime){...}
2 + static boolean isAppXml(String mime){...}
3 + static boolean isTestXml{String mime}{...}
```

(b) Partial changes of XmlStreamReader.java from Commit d46782 of Commons IO **(NIT)**

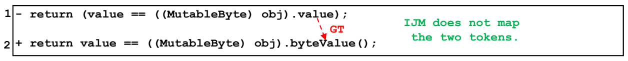


```
    public void testFilterSet(){
1 -   IOFileFilter filter =  FileFilterUtils.nameFileFilter("A");
2 +   final IOFileFilter filter = FileFilterUtils.nameFileFilter("A");
    }

    public void testFilterList_fromArray() throws Exception {
3 +   final IOFileFilter filter = FileFilterUtils.nameFileFilter("A");
    }
```

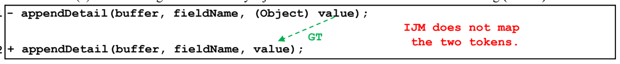
(c) Partial changes of FileFilterTestCase.java from commit 6aa007 of **(PM)**

Fig. 5: Illustrative examples of inaccurately mapped statements that can be identified using our measures.



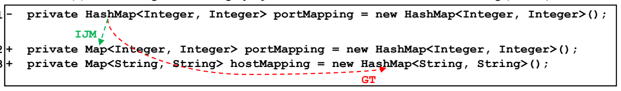
```
1 - return (value == ((MutableByte) obj).value);
2 + return value == ((MutableByte) obj).byteValue();
```

(a) Partial changes of MutableByte.java from commit 49e1f1 of Commons Lang **(TYPE)**



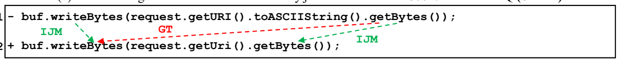
```
1 - appendDetail(buffer, fieldName, (Object) value);
2 + appendDetail(buffer, fieldName, value);
```

(b) Partial changes of ToStringStyle.java from commit 85d334 of Commons Lang **(STMT)**



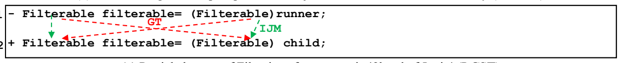
```
1 -  private HashMap<Integer, Integer> portMapping = new HashMap<Integer, Integer>();
2 +  private Map<Integer, Integer> portMapping = new HashMap<Integer, Integer>();
3 +  private Map<String, String> hostMapping = new HashMap<String, String>();
```

(c) Partial changes of PublishedAddressPolicy.java from commit 4800a7 of ActiveMQ **(STMT)**



```
1 - buf.writeBytes(request.getURI().toASCIIString().getBytes());
2 + buf.writeBytes(request.getUri().getBytes());
```

(d) Partial changes of HttpRequestEncoder.java from commit 2d9277 of Netty **(VALUE)**



```
1 - Filterable filterable= (Filterable)runner;
2 + Filterable filterable= (Filterable) child;
```

(e) Partial changes of Filter.java from commit 42beed of Junit4 **(LCST)**

Fig. 6: Illustrative examples of inaccurately mapped tokens that can be identified using our measures.

PM characterizes whether the parent nodes of a pair of mapped statements are also mapped. We observe that statements with mapped parent nodes are more likely to be mapped. For a pair of mapped statements, we check if mapping one of the statements to another statement with mapped parent nodes can achieve the same NIT. For instance, in Fig. 5(c), IJM maps the statements at lines 1 and 2 with mapped parent nodes, while GT maps the statements at lines 1 and 3 with parent nodes not mapped. We determine that IJM is more accurate than GT in mapping the statement at line 1. Furthermore, we notice a special type of statements, i.e., blocks. A block is a group of statements between balanced braces (i.e., "{" and "}"). We observe that a block should be mapped along with its parent nodes, e.g., the "{" following the method `testFilterSet` in Fig. 5(c) should be mapped along with the method declaration. Thus, mapped blocks with unmapped parent nodes are determined to be inaccurate.

TYPE characterizes whether mapped tokens have the same type. For a token whose directly relevant node is not a name node, we define the type of the token as the label of its directly relevant node. For tokens whose directly relevant node is a name node, we define four types: variable name, type name, method name and declaration name. Following Frick et al. [12], we consider the mapping of tokens with different types as inaccurate. For instance, in Fig. 6(a), GT maps a variable name `value` to a method name `bytevalue`, we determine that such a mapping is inaccurate.

STMT characterizes whether mapped tokens belong to a pair of mapped statements. Two tokens from mapped statements are more likely to be mapped. We observe that mapping tokens from mapped statements is better than (1) not mapping the tokens and (2) mapping tokens from unmapped statements. For instance, in Fig. 6(b), the two `value` tokens are both variable names and they belong to a pair of mapped statements. GT maps the two tokens, while IJM does not map them. We determine that GT is more accurate than IJM in mapping the two tokens. As another example, we find that both GT and IJM map the statement at line 1 to the statement at line 2 in Fig. 6(c). GT maps the two `HashMap` tokens in unmapped statements, while IJM maps the `HashMap` to the `Map` from the mapped statements. Using the STMT measure, we determine that IJM is more accurate in mapping the `HashMap` at line 1 than GT.

VAL characterizes whether mapped tokens have the same string value. Two identical tokens in a pair of mapped statements are more likely to be mapped. We consider that mapping such two tokens is better than mapping one of the tokens to another token with different values between the two statements. For instance, in Fig. 6(d), GT maps `getBytes` to `writeBytes`, while IJM maps the two `getBytes` tokens. Using the VAL measure, we determine that IJM is more accurate in mapping the `getBytes` tokens than GT.

LLCS is defined as the length of the longest common subsequence (LCS) [16] that is calculated using the mapped tokens between mapped statements. We observe that the order of tokens is infrequently changed in a statement. We use LLCS to quantify the number of tokens that are sequentially mapped between the mapped statements. For instance, in

Fig. 6(e), GT changes the orders of the two `Filterable` tokens in the statement at line 1. As a result, at most three tokens are sequentially mapped between the statements, i.e., `Filterable`, `=` and `runner`. The LLCS for the mapped tokens is calculated as three. IJM sequentially maps the five tokens between the two statements. The LLCS for the mapped tokens is calculated as five. Using the LLCS measure, we determine that IJM is more accurate in mapping the `Filterable` tokens than GT.

Table II shows the number of statements that are categorized along the measures. The measures can identify 157 (85%), 208 (86%) and 178 (86%) of the statements with inaccurate mappings for GT, MTD and IJM, respectively. The other statements with inaccurate mappings are categorized into the **Other** category. For these statements, we find that determining the accuracy of the mappings of statements and tokens requires more comprehension of the changes.

Usage of the similarity measures. We compare the generated mappings by two algorithms using the following steps.

Step 1. If an algorithm maps two non-block statements with an NIT of 0, we determine the mapping as inaccurate. If an algorithm maps two blocks with unmapped parent nodes, we determine the mapping as inaccurate. If an algorithm maps two tokens with different types, we determine the mapping as inaccurate.

Step 2. When the two algorithms map a statement to different statements, mapping statements with a larger NIT is considered to be more accurate. If the two pairs of statements have the same NIT, mapping statements with mapped parent nodes is considered to be more accurate than mapping statements with unmapped parent nodes.

Step 3. When the two algorithms consistently map a statement, we assume that the two algorithms accurately map the statement. From the statement, we retrieve all the tokens that are inconsistently mapped by the two algorithms. For a token that is inconsistently mapped by the two algorithms, we first use the STMT measure to compare the generated mappings for the token by the two algorithms. If both algorithms map tokens from the mapped statements, mapping identical tokens is considered to be more accurate than mapping tokens with different values. If both algorithms map identical tokens from the mapped statements, the mapped tokens with larger LLCS are considered to be more accurate.

VI. Evaluation

We evaluate our approach by answering three research questions. In this section, we present the three research questions and our answer to each question.

A. (RQ1) How effective is our approach in detecting statements with inaccurate mappings for the studied algorithms?

Motivation. By answering this research question, we investigate if our approach can effectively find the statements with inaccurate mappings as generated by the studied algorithms.

Method. Our approach may be over-fit on the used dataset in our manual analysis. Thus, we conduct an experiment with 12 external experts. The experts include PhD students and post-doctors majoring in software engineering. They have three to seven years of programming experience in Java. Seven experts have prior experience working in industry. For each project, we randomly select 20 statements from all the file revisions. For each selected statement, at least two studied algorithms inconsistently map the statement or its tokens. In total, we select 200 of such statements with inconsistent mappings. The selected statements involve various change patterns including adding, deleting, moving and updating statements and tokens.

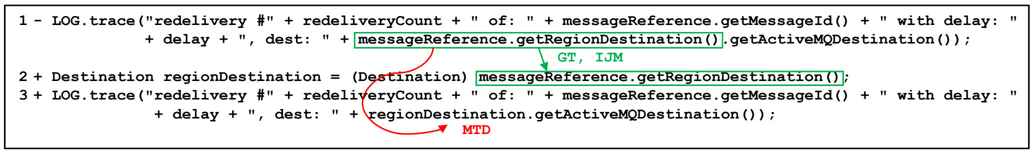
We randomly divide the 200 statements into four groups with each group having 50 statements. We also divide the experts into four groups with each group having three experts. We invite the four groups of experts to analyze the four groups of statements, respectively. For each statement, we provide the mappings of the statement and its tokens as generated by each of the studied algorithms. Notice that we do not provide the algorithm that generates the mappings. We let the experts determine if the mapping of the statement or a token of the statement is inaccurate. For each group of statements, we calculate Fleiss' Kappa [10] to estimate the agreement of the three experts' determination results.

For each studied algorithm, we have three determination results on the accuracy of algorithm in mapping each statement and its tokens. For each statement and the generated mappings of the statement and its tokens by an algorithm, we label the mappings as inaccurate if at least two experts determine that inaccurate mappings exist.

Then, we run our approach to determine statements with inaccurate mappings for GT, MTD and IJM from the 200 statements. Finally, we compare the determination results of our approach with the experts' determination results. We define a true positive as a statement with inaccurate mappings for an algorithm that is determined as such by both our approach and experts. We define a false positive as a statement with inaccurate mappings for an algorithm that is determined as such by our approach but not determined as such by experts. We define a false negative as a statement with inaccurate mappings for an algorithm that is determined as such by experts but not determined as such by our approach. Let us denote the number of true positives, false positives and false negatives as TP, FP and FN. We calculate the precision of our approach as $\frac{TP}{TP+FP}$. And we calculate the recall of our approach as $\frac{TP}{TP+FN}$.

Results. For the four groups of statements, the Kappa values for the experts' determination results are 0.81, 0.82, 0.84 and 0.78, respectively. Thus, the experts' determination results have an excellent agreement.

Table III presents the TP, FP, FN, precision and recall of our approach in determining statements with inaccurate mappings for the studied algorithms. As shown in the table, our approach achieves a precision of 0.98–1.00 and a recall of 0.65–0.75. Almost all of the statements with inaccurate mappings as determined by our approach are also determined as such by



```
1 - LOG.trace("redelivery #" + redeliveryCount + " of: " + messageReference.getMessageId() + " with delay: "
             + delay + ", dest: " + messageReference.getRegionDestination().getActiveMQDestination());

2 + Destination regionDestination = (Destination) messageReference.getRegionDestination();
3 + LOG.trace("redelivery #" + redeliveryCount + " of: " + messageReference.getMessageId() + " with delay: "
              + delay + ", dest: " + regionDestination.getActiveMQDestination());
```

Fig. 7: GT and IJM generates accurate mappings but our approach determines the mapping of a token as inaccurate.

TABLE III: TP, FP, FN, precision and recall of our approach.

Alg.	TP	FP	FN	Precision	Recall
GT	56	1	27	0.98	0.67
MTD	90	0	30	1.00	0.75
IJM	59	1	32	0.98	0.65

experts.

For the false positives and false negatives, we further asked the experts why they considered that an algorithm inaccurately maps a statement or tokens of the statement. We analyze cases of false positives and false negatives. For the two false positives, we find that GT and IJM generate the accurate mappings for a statement and its tokens but our approach determines the mapping of a token as inaccurate. We show this case in Fig. 7. As shown in Fig. 7, the code involves a refactoring that extracts the method invocation `messageReference.getRegionDestination` in the statement at line 1 as a new variable. GT and IJM accurately map the invocation to the statement at line 2, while MTD maps `messageReference` in the statement at line 1 to the `regionDestination` in the statement at line 3. In this case, mapping tokens from unmapped statements is better than mapping tokens from mapped statements. However, when comparing the generated mappings by GT, IJM and MTD, our approach considers that MTD generates a better mapping than GT and IJM. This case indicates that our approach can be further improved by considering refactoring changes.

For 11 cases of false negatives, we find that our similarity measures can distinguish the accurate and inaccurate mappings of statements or tokens. However, all three algorithms generate inaccurate mappings. Hence, the inaccurate mappings cannot be detected by comparing the similarity measures of the mapped statements and tokens between different algorithms. We observe 38 cases where an algorithm maps two tokens from unmapped statements and another algorithm separately maps the two tokens to empty elements. We observe 33 cases where an algorithm maps two statements and another algorithm separately maps the two statements to empty elements. We further observe 7 cases where two algorithms map a statement or token to different statements or tokens but our similarity measures cannot distinguish accurate and inaccurate mappings. In these 78 cases, determining the inaccurate algorithm requires more syntactic information to determine if mapping two tokens or two statements helps understand the changes.

Summary. Our approach achieves a precision of 0.98–1.00 and a recall of 0.65–0.75 in determining the statements with inaccurate mappings for the studied algorithms. Any statements with inaccurate mappings that we detect are highly likely to be correct, although there may be additional inaccurate mappings that we cannot detect. Our approach can be used to estimate the lower bound on the effectiveness of AST mapping algorithms.

TABLE IV: TP, FP, FN, precision and recall of our approach when comparing an algorithm with another algorithm.

Comparison	Alg.	TP	FP	FN	Precision	Recall
GT vs. MTD	GT	44	1	39	0.98	0.53
	MTD	83	0	37	1.00	0.69
GT vs. IJM	GT	47	0	36	1.00	0.57
	IJM	51	0	40	1.00	0.56
MTD vs. IJM	MTD	74	0	46	1.00	0.62
	IJM	49	1	42	0.98	0.54

B. (RQ2) How effective is our approach when comparing an algorithm with multiple algorithms than when comparing it with another algorithm?

Motivation. As described in Section V-E, we separately compare an algorithm with the other two algorithms. Then, we calculate a union set of statements with inaccurate mappings for the algorithm. We investigate if comparing an algorithm with the other two algorithms is more effective in detecting statements with inaccurate mappings than comparing it with another algorithm.

Method. We have three pairs of studied algorithms, namely GT vs. MTD, GT vs. IJM and MTD vs. IJM. For the 200 analyzed statements in RQ1, we use our approach to compare the generated mappings for statements and tokens by each pair of algorithms. For each pair of algorithms, we calculate a set of statements with inaccurate mappings. In such a case, we compare an algorithm with another algorithm. Then, we calculate the precision and recall of our approach in detecting statements with inaccurate mappings for the two algorithms. The precision and recall of our approach that compares an algorithm with the other two algorithms are shown in Table III. Finally, we compare the results shown in the table with the precision and recall of our approach that compares an algorithm with another algorithm.

Results. Table IV presents TP, FP, FN, precision and recall of our approach when comparing an algorithm with another algorithm. By comparing the results show in Tables III and IV, we find that our approach achieves a better recall with a difference of 9%–23% when comparing an algorithm with two algorithms than when comparing it with another algorithm. On the other hand, the precision of our approach is not impacted.

TABLE V: Number of statements and file revisions for which the studied algorithms are determined to generate inaccurate mappings.

Projects	Statements			File Revisions		
	GT	MTD	IJM	GT	MTD	IJM
ActiveMQ	53,083	191,566	39,669	5,817	8,523	5,786
Commons IO	7,932	16,978	5,883	546	713	656
Commons Lang	23,306	53,533	21,567	1,501	1,823	1,641
Commons Math	43,450	101,194	34,440	3,703	4,588	3,881
Junit4	6,997	13,800	5,750	924	1,083	984
Hibernate ORM	127,770	356,412	93,146	13,026	16,414	13,069
Hibernate Search	43,942	112,419	38,919	6,012	7,326	6,964
Spring Framework	164,545	440,480	145,374	17,120	20,484	20,215
Spring Roo	54,245	172,454	38,397	4,573	5,764	4,963
Netty	130,249	374,883	91,795	11,584	14,191	11,451

As described in Section V-F, two algorithms may generate the same mapping that is inaccurate. Such an inaccurate mapping cannot be detected by comparing the two algorithms. If another algorithm generates the accurate mapping, comparing the third algorithm with the former two algorithms may reveal the inaccurate mapping. Thus, comparing an algorithm with multiple algorithms can detect more inaccurate mappings.

Summary. Our approach can detect 9%–23% more statements with inaccurate mappings when comparing an algorithm with the other two algorithms than when comparing it with another algorithm.

C. (RQ3) Do state-of-the-art AST mapping algorithms generate many inaccurate mappings?

Motivation. We show that our approach achieves a nearly perfect precision in finding statements with inaccurate mappings for the studied algorithms. Hence, we leverage our approach to investigate whether the studied algorithms generate many inaccurate mappings.

Method. We leverage GT, MTD and IJM to calculate the mappings of the AST nodes for all the file revisions of the ten studied projects. For each file revision, we use our approach to detect the statements with inaccurate mappings for each studied algorithm. For each project, we count the detected statements with inaccurate mappings for each algorithm. We also count the file revisions for which the studied algorithms are determined to generate inaccurate mappings.

Results. Table V presents the number of statements with inaccurate mappings as detected by our approach. We also show the number of file revisions for which the studied algorithms are determined to generate inaccurate mappings. As shown in the table, the three studied algorithms may generate a considerable number of inaccurate mappings. For each project, we further calculate the ratio of file revisions for which the studied algorithms are determined to generate inaccurate mappings. We find that GT, MTD and IJM are determined to generate inaccurate mappings for 20%–29%, 25%–36% and 21%–30% of the file revisions, respectively.

Summary. GT, MTD and IJM are determined to generate inaccurate mappings for a considerable number of file revisions. State-of-the-art AST mapping algorithms still have room for improvement.

VII. Discussion

A. Threats to Validity

The primary threats to the validity of our experiments are twofold. First, we compare the determination results of our approach and experts on the accuracy of generated mappings by the studied algorithms for 200 statements. The number of analyzed statements is not very large-scale. This is because such a manual analysis is time-consuming, with understanding mappings of each statement and each token. On average, each expert takes 1.5 hours to analyze the allocated 50 statements. The 200 statements are randomly taken from 10 different projects, and they are from different file revisions. Dotzler et al. analyzed only 10 file revisions when evaluating MTD [8]. Our analysis involves much more file revisions than their analysis. Second, when we select the statements, we require that at least two studied algorithms inconsistently map the statement or its tokens. There may exist cases where the studied algorithms consistently map a statement and its tokens, but the mapping of the statement or a token of the statement is inaccurate. The selected statements do not consider such cases, and our approach cannot detect the inaccurate mapping in such cases. We manually analyzed 100 statements for which the studied algorithms generate consistent mappings at both statement and token levels. We did not observe the cases where the three algorithms produce inaccurate mappings. Nevertheless, our code and data are made publicly available [1], and researchers are encouraged to investigate this possibility.

B. Limitations

From our experiments, we observe two limitations of our approach. First, as described in our answer to RQ1, refactoring changes may impact the precision of our approach. In refactoring changes, mapping tokens from unmapped statements may be better than mapping tokens from mapped statements. We note that researchers proposed several refactoring detection tools, e.g., [30]. Incorporating such tools into our approach may deal with this limitation. On the other hand, there still exists a considerable number of inaccurate mappings that cannot be detected by our approach. According to our answer to RQ2, comparing an algorithm with more algorithms may detect more inaccurate mappings as generated by the algorithm. Moreover, researchers have proposed various heuristics to map program elements [20]. Additional similarity measures can be derived from these heuristics. Such measures may further improve the recall of our approach. Our code and data are made publicly available [1], and researchers are encouraged to extend our approach.

VIII. Related Work

A. AST mapping algorithms

Many AST mapping algorithms were proposed in prior studies. Yang proposed an AST mapping algorithm using a branch-and-bound implementation of the largest common subtree problem [33]. This algorithm does not consider moved AST nodes. Fluri et al. proposed ChangeDistiller, an AST mapping algorithm that uses a reduced AST, in which code statements are encoded as leaf nodes [11]. Hashimoto et al. proposed Diff/TS, an algorithm that works with raw ASTs and supports multiple languages [14]. Nguyen et al. proposed Jsync, which leverages a classic text-based mapping algorithm to map AST nodes [28]. Recently, researchers proposed GumTree [9], MTDiff [8] and IJM [12]. These algorithms are the state-of-the-art AST mapping algorithms and are analyzed in our paper. Different from them, we focus on evaluating AST mapping algorithms rather than proposing a new AST mapping algorithm.

B. Use of AST mapping algorithms

AST mapping algorithms are widely used in several SE research areas. ChangeDistiller has been used to identify non-essential modifications [19] and automate repetitive edits [26]. Nguyen et al. used Jsync to track cloned code in the software evolution process [28]. Moreover, many studies used GumTree to analyze code patterns of changes such as bug-fixing changes [5], [13], [18], [21], [23], [29], logging changes [22] and changes to online code examples [34]. Also, prior work trained models based on the edit actions of changes that are calculated using GumTree [7], [15], [24], [31], [32]. Such models are used to recommend changes such as patches [32] and logging changes [22]. Different from them, we focus on evaluating AST mapping algorithms instead of using the algorithms to analyze changes.

IX. Conclusion and Future Work

In this paper, we propose a differential testing approach that can automatically determine the statements with inaccurate mappings for AST mapping algorithms. Given a file revision, we first compare the generated mappings by different algorithms and extract the statements with inconsistent mappings. Then, we use six similarity measures collectively to compare the mapped statements and tokens by different algorithms. By doing so, we determine the statements with inaccurate mappings for each of the algorithms.

By conducting an experiment with 12 experts, we show that our approach achieves a precision of 0.98–1.00 and a recall of 0.65–0.75. The studied algorithms are determined to generate inaccurate mappings for a considerable number (20%–36%) of file revisions in our studied projects. Hence, state-of-the-art AST mapping algorithms still need improvements. AST mapping algorithms play a foundational role in many existing studies. It is necessary to investigate if the inaccurate mappings as generated by the algorithms impact the conclusions of existing studies.

Acknowledgement

This research was partially supported by the National Key R&D Program of China (No. 2019YFB1600700), Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100021), ARC Discovery grant (DP200100020), Key Research and Development Program of Zhejing Province (No. 2021C01014), and the National Research Foundation, Sinapore under its Industry Alignment Fund - Prepositioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] Code and data of this paper. http://doi.org/10.5281/zenodo.4281091.

[2] Github repository of gumtree. https://github.com/GumTreeDiff/gumtree.

[3] Github repository of ijm. https://github.com/VeitFrick/IJM.

[4] Github repository of mtdiff. https://github.com/FAU-Inf2/treedifferencing.

[5] E. C. Campos and M. d. A. Maia. Discovering common bug-fix patterns: A large-scale observational study. *Journal of Software: Evolution and Process*, 31(7):e2173, 2019.

[6] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *Acm Sigmod Record*, 25(2):493–504, 1996.

[7] B. Danglot, M. Monperrus, W. Rudametkin, and B. Baudry. An approach and benchmark to detect behavioral changes of commits in continuous integration. *Empirical Software Engineering*, pages 1–37, 2020.

[8] G. Dotzler and M. Philippsen. Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–671. IEEE, 2016.

[9] 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*, pages 313–324, 2014.

[10] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[11] 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*, 33(11):725–743, 2007.

[12] V. Frick, T. Grassauer, F. Beck, and M. Pinzger. Generating accurate and compact edit scripts using tree differencing. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 264–274. IEEE, 2018.

[13] Q. Hanam, F. S. d. M. Brito, and A. Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–156, 2016.

[14] M. Hashimoto and A. Mori. Diff/ts: A tool for fine-grained structural change analysis. In *2008 15th working conference on reverse engineering*, pages 279–288. IEEE, 2008.

[15] F. Hassan and X. Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1078–1089. IEEE, 2018.

[16] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.

[17] 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*, pages 679–690, 2018.

[18] M. R. Islam and M. F. Zibran. How bugs are fixed: exposing bug-fix patterns with edits and nesting levels. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1523–1531, 2020.

[19] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 351–360. IEEE, 2011.

[20] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64, 2006.

[21] 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*, pages 1–45, 2020.

[22] S. Li, X. Niu, Z. Jia, J. Wang, H. He, and T. Wang. Logtracker: learning log revision behaviors proactively from software evolution history. In *Proceedings of the 26th Conference on Program Comprehension*, pages 178–188, 2018.

[23] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon. A closer look at real-world patches. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 275–286. IEEE, 2018.

[24] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security*, pages 229–246. Springer, 2017.

[25] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[26] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 502–511. IEEE, 2013.

[27] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522, 2016.

[28] 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*, 38(5):1008–1026, 2011.

[29] Z. Ni, B. Li, X. Sun, T. Chen, B. Tang, and X. Shi. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software*, 163:110538, 2020.

[30] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 483–494. IEEE, 2018.

[31] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–312. IEEE, 2018.

[32] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[33] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[34] T. Zhang, D. Yang, C. Lopes, and M. Kim. Analyzing and supporting adaptation of online code examples. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 316–327. IEEE, 2019.