iASTMapper: An Iterative Similarity-Based Abstract Syntax Tree Mapping Algorithm

Neng Zhang
Sun Yat-sen University
China
zhangn279@mail.sysu.edu.cn

Qinde Chen*
Sun Yat-sen University
China
chenqd6@mail2.sysu.edu.cn

Zibin Zheng
Sun Yat-sen University
China
zhzibin@mail.sysu.edu.cn

Ying Zou
Queen's University
Canada
ying.zou@queensu.ca

***Abstract*—Abstract syntax tree (AST) mapping algorithms are widely used to locate the code changes in a file revision by mapping the AST nodes of the source code before and after the code changes. A recent differential testing of three state-of-the-art AST mapping algorithms, i.e., GumTree, MTDiff, and IJM, reveals that the algorithms generate inaccurate mappings for a considerable number of file revisions. We find that the inaccurate mappings could be caused by the mutual influence: the mappings of lower-level AST nodes (e.g., tokens) have impacts on the mappings of higher-level AST nodes (e.g., statements) and vice versa. This mutual influence issue is rarely considered by existing algorithms.**

In this paper, we propose an algorithm, called iASTMapper, that iteratively map two ASTs based on the similarities between AST nodes. Given a file revision, we extract three types of AST nodes in different levels of program structures (i.e., tokens, statements, and inner-statements) from the ASTs of the two source code files. We first build mappings of the unchanged statements and inner-statements. Then, we use an iterative method to map the rest of the nodes without mapping. For each of the three types of nodes, we iteratively map the nodes based on their similarities measured using heuristic rules. We further use an iterative mechanism to connect the three iterative mapping processes by considering the mutual influence between the mappings of different types of nodes. Finally, a series of code edit actions are generated from the node mappings to help users understand and locate the code changes during revisions. We conduct experiments to compare iASTMapper with three baselines, i.e., GumTree, MTDiff, and IJM, by automatically evaluating 210,997 file revisions from ten Java projects. Furthermore, we manually evaluate the correctness of the code edit actions generated for 200 file revisions with 12 evaluators. The results demonstrate that iASTMapper outperforms the baselines. iASTMapper can generate shorter code edit actions by at least 1.29% than the baselines, with a high accuracy of 96.23%.

***Index Terms*—AST mapping, Code change analysis**

I. Introduction

An abstract syntax tree (AST) mapping algorithm is used to map the program elements (e.g., statements and tokens) between two versions of a source code file, i.e., a file revision. The algorithm converts the source code files before and after a revision to ASTs and generate mappings of the nodes between the ASTs by estimating the similarities of the nodes. Based on the mappings, a series of AST edit actions can be generated to describe the code changes in the file revision in terms of AST nodes. The edit actions can be used for many downstream program analysis tasks, such as mining code change patterns [33] and automatic program repair [36].

Several AST mapping algorithms are proposed by researchers, e.g., GumTree (GT) [2], MTDiff (MTD) [3] and IJM [4]. These algorithms aim to 1) maximize the number of mapped AST nodes and 2) minimize the number of AST edit actions. The number of mapped AST nodes can reflect the mapping capability of an algorithm. If an algorithm generates more AST node mappings, it has a higher probability of generating more accurate AST edit actions. The smaller number of AST edit actions is, the less efforts are required by users to understand the code changes. However, Fan et al. [5] reveal that the three widely used AST mapping algorithms, i.e., GT, MTD, and IJM, generate inaccurate mappings for 20%-36% of the file revisions from ten Java projects. We further perform an in-depth analysis of the inaccurate mappings and find that the mappings of AST nodes in different levels of program structures have mutual influence: the mappings of lower-level nodes (e.g., tokens) can affect the mappings of higher-level nodes (e.g., statements) and vice versa. This mutual influence issue is rarely considered in the design of the algorithms. Moreover, the description of AST edit actions requires users to manually track the AST nodes involved in the actions. Thus, AST edit actions might be challenging for users to use.

In this paper, we propose an algorithm, called iASTMapper, that can iteratively map AST nodes based on the similarities of nodes among two versions of an AST. Given a file revision, we obtain the ASTs of both source code files and extract three types of AST nodes in different structural levels: statements, inner-statements, and tokens. A detailed introduction of these nodes are presented in Section III-A. Then, we perform a fast mapping of unchanged statements and inner-statements between the two ASTs. After that, an iterative method is used to associate the rest of nodes that are not mapped in the prior step. For each of the three types of nodes, we iteratively map the nodes in the same level by measuring

their similarities using a number of heuristic rules. We use an iterative mapping mechanism for each type of node as we observe that several rounds are needed to obtain a stable set of mappings. Furthermore, we build another iterative mechanism by connecting the three iterative mapping mechanisms. This iterative mechanism is built based on the mutual influence between the mappings of different levels of nodes. A series of code edit actions are finally generated from the node mappings to describe the code changes in terms of source code elements. Compared with the traditional AST edit actions, code edit actions are easier for users to read and understand without the need of tracking the ASTs.

To evaluate iASTMapper, we collect 210,997 file revisions of ten Java projects from GitHub. We conduct a large-scale automatic evaluation of iASTMapper using state-of-the-art baselines: GT, MTD, and IJM. iASTMapper increases the number of AST node mappings by at least 25.18% compared to the baselines and generates the shortest code edit scripts. Moreover, 12 evaluators are recruited to assess the accuracy of the code edit actions generated for 200 file revisions[1]. iASTMapper achieves an accuracy of 96.23%, which outperforms GT, IJM, and MTD by 4.66%, 7.29%, and 24.62%, respectively. We also validate the importance of the iterative mechanisms applied in iASTMapper.

The contributions of this work are outlined below:

- We propose an iterative similarity-based AST mapping algorithm, iASTMapper. It iteratively build the mappings of three types of AST nodes using the similarities of nodes. We propose heuristic rules for similarity measurements.
- We conduct a large-scale automatic evaluation and a manual evaluation, which demonstrate the superiority of iASTMapper to state-of-the-art AST mapping algorithms.

It is worthy to note that determining the optimal mapping between two ASTs is a NP-hard problem. A trade-off exists between the accuracy and efficiency of an AST mapping algorithm. iASTMapper employs a number of heuristic rules and an iterative algorithm to enhance the accuracy of AST node mappings. However, it comes at the cost of efficiency, especially when processing file revisions with large and/or complex ASTs. Moreover, we elucidate the iASTMapper procedure for analyzing Java file revisions. When applying iASTMapper to file revisions written in other languages, two configurations are necessary, namely 1) the selection of a tool capable of generating ASTs for the revisions; and 2) the definition of the three types of nodes (i.e., statements, inner-statements, and tokens) in the ASTs.

II. Preliminaries

Abstract Syntax Tree (AST). An AST is parsed from a source code file to represent the source code in a tree structure. It is composed of a set of nodes and a set of edges that represent parent-child relationships between nodes. A node n_1 is the *parent* of another node n_2 if n_2 is a child of n_1. The node

[1]Our evaluation complies with the ACM Policy on Research Involving Humans.

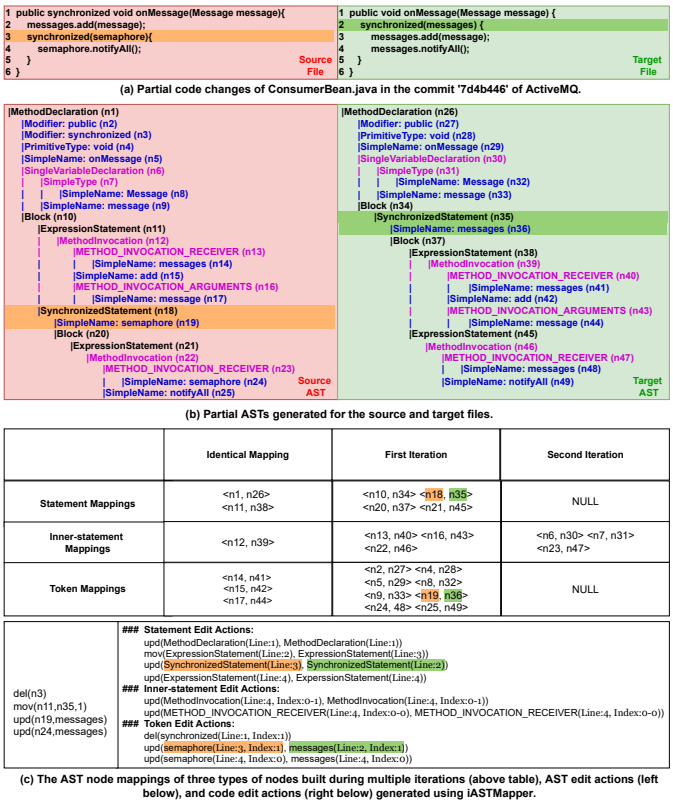

Fig. 1. Example of the ASTs, AST node mappings, AST edit actions, and code edit actions generated by iASTMapper.

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 (connected by edges) to the root node are called its *ancestors*, and the node is called their *descendant*. Each node represents a code element with a *label* (e.g., `MethodDeclaration`) to indicate its type. Some nodes have a *value* (e.g., `public`) to indicate the corresponding tokens of the element.

AST mapping algorithms. An AST mapping algorithm maps the nodes between two ASTs: a *source AST* parsed from the source code file before code changes (referred to as *source file*) and a *target AST* parsed from the source code file after code changes (referred to as *target file*). The key is to measure the similarities between the nodes from the two ASTs to determine whether two nodes can be mapped or not. Only the nodes with the same label can be mapped. A node mapping can be represented as $\langle n_s, n_t \rangle$, where n_s is a node in the source AST, and n_t is a node in the target AST.

AST edit actions. A series of AST edit actions can be generated from the mappings of AST nodes to describe the process of transforming the source AST to the target AST. Existing AST mapping algorithms, e.g., GT, MTD, and IJM, often use Chawathe et al.'s algorithm [12] to calculate the AST edit actions. There are four types of AST edit actions:

- $upd(n, v)$ replaces the value of the node n with v.
- $add(n, p, i)$ adds a node n as the i^{th} child of the node p if p is not null. Otherwise, n is added as the new root

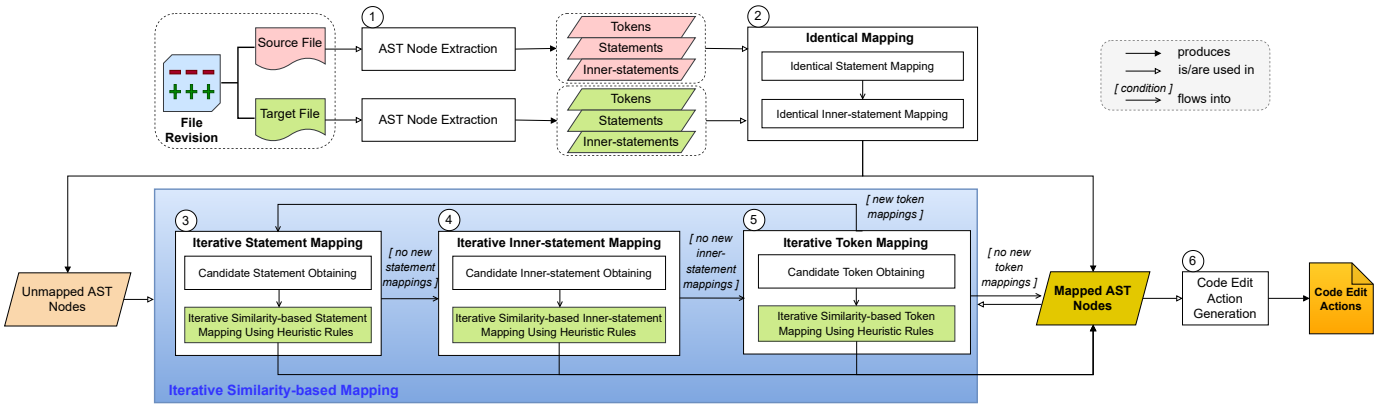

Fig. 2. An overview of the framework of iASTMapper.

node.

- $del(n)$ deletes the leaf node n.
- $mov(n, p, i)$ moves the node n with the subtree rooted at n as the i^{th} child of the node p.

Fig. 1(a) shows partial code changes of ConsumerBean.java from the commit '7d4b446' of the Java project ActiveMQ. Fig. 1(b) shows the two ASTs generated for the source code before and after the code changes. In Fig. 1(c), the above table presents the mappings of three types of AST nodes generated during the identical mapping step and the iterative mapping process by iASTMapper. The two synchronized statements, i.e., n_{18} in the source AST and n_{35} in the target AST, are mapped in the first iteration. The left below part of Fig. 1(c) lists the four AST edit actions, e.g., $del(n_3)$, generated from the AST node mappings using Chawathe et al's algorithm.

III. Approach

Figure 2 shows the overview framework of iASTMapper. Given a file revision that contains two source code files: a *source file* before code changes and a *target file* after code changes, iASTMapper generates the AST node mappings and code edit actions using six steps ① - ⑥. For each source code file, we generate its AST and extract three types of nodes: *statements*, *inner-statements*, and *tokens*. Then, we perform a mapping of the statements and inner-statements identical in both ASTs. Next, the unmapped nodes are mapped using an iterative similarity-based method. Finally, a series of code edit actions are generated from the mappings to describe the code changes happened in the file revision.

A. AST Node Extraction

For each of the source file and the target file of a Java file revision, we generate a standard AST using the Eclipse JDT parser and extract three types of AST nodes as follows.

Tokens. We extract tokens from the AST nodes that have a value. We ignore the AST nodes representing comments and Javadocs as they are typically not treated as code. Given the source AST shown in Fig. 1(b), 12 tokens are extracted, i.e., the nodes marked in blue, e.g., n_2-n_5.

Statements. We extract statements from the AST nodes corresponding to a statement in the source code. In Java, we identify two types of statements: 1) a statement ends with a semicolon ';', e.g., a variable declaration statement; and 2) a statement ends with a left curly brace '{', e.g., a method declaration statement. Given the source AST shown in Fig. 1(b), six statements are extracted, i.e., the nodes marked in black, e.g., n_1. The Block nodes, e.g., n_{10}, are viewed as a specific type of statement. Note that a statement may have descendant statements, e.g., a method declaration statement may contain expression statements. We obtain the *value* of a statement by grouping the values of its descendant nodes (except the descendant statements). The value of n_1 is '*public synchronized void onMessage Message message*'.

Inner-statements. We extract inner-statements from the AST nodes under an AST node corresponding to a statement, except the AST nodes corresponding to tokens. The *value* of an inner-statement is obtained by grouping the values of its descendant nodes. Given the source AST shown in Fig. 1(b), seven inner-statements are extracted, i.e., the nodes marked in magenta, e.g., n_6. The value of n_6 is '*Message message*'.

We name the three types of nodes using their corresponding program elements in source code. When mentioning a type of node or element, we do not explicitly express them, e.g., token nodes, if there is no ambiguity based on the context.

B. Identical Mapping

We analyze the changed code lines of 210,997 file revisions from ten Java projects and find that in total there are only 4.43% of the code lines changed, as listed in Table II. The large amount of unchanged code can be easily mapped between the two versions of ASTs and thus greatly reduce the amount of code that needs to be mapped using our iterative similarity-based method (see Section III-C). We use the following two substeps to map the unchanged statements and inner-statements extracted from the source and target ASTs.

*1) **Identical Statement Mapping**:* Two statements are identical if they have the same value. In particular, in a Java file, each method has a unique signature that includes the method name and the list of parameter types. Therefore, for

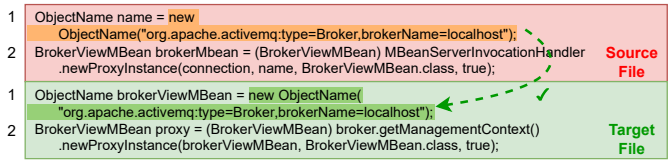

Fig. 3. An identical inner-statement mapping generated for partial code changes of AdvisoryJmxTest.java from the commit '0683d8b' of ActiveMQ.

two method declaration statements (e.g., the nodes n_1 and n_{26} shown in Fig. 1(b)), they are identical if they have the same signature. We map a statement, s_i, from the source AST to a statement, s_j, from the target AST if 1) s_i and s_j are identical; 2) s_i has no other identical statements except s_j in the target AST; and 3) s_j has no other identical statements except s_i in the source AST. That is, we require a one-to-one mapping between two identical statements to ensure a high accuracy of mapping. For a pair of mapped statements, we further map the tokens (*resp.* inner-statements) at the same position (*resp.* position range) in both statements. If a statement has multiple identical statements, then it is hard to determine which identical statement should be mapped to the statement, without additional context information. We subsequently use an iterative similarity-based method to address this problem.

*2) **Identical Inner-statement Mapping**:* In spite of the inner-statements mapped above, we map other identical inner-statements that have the same value but belong to non-identical statements, as shown in Fig. 3. We also require a one-to-one mapping between two identical inner-statements to ensure a high accuracy of mapping. Since there can be a large number of inner-statements along the path from each statement to the leaf tokens, it is time-consuming to compare and map all inner-statements. We can map the relatively long inner-statements that have no less than δ descendant tokens. In this work, we set δ = 5. For a pair of mapped inner-statements, we further map the tokens (*resp.* inner-statements) at the same position (*resp.* position range) in both inner-statements.

We randomly sample 383 identical statement mappings and 383 identical inner-statement mappings generated for the file revisions from ten projects, which are statistically significant sample sizes considering a confidence level of 95% and a confidence interval of 5%. The first two co-authors independently examine the correctness of each mapping and label the correct mappings with '1', otherwise '0'. From both co-authors' results, the mappings are all correct, indicating that both types of identical mappings have a high accuracy.

C. Iterative Similarity-based Mapping

After the identical mapping step, we obtain a set of pairs of mapped nodes, $\mathcal{M}$, a set of unmapped nodes from the source AST, $\mathcal{UM}_s$, and a set of unmapped nodes from the target AST, $\mathcal{UM}_t$. We use an iterative similarity-based method to map the nodes in $\mathcal{UM}_s$ and $\mathcal{UM}_t$. As shown in Fig. 2, the method has an iterative mapping mechanism for each of the three types of

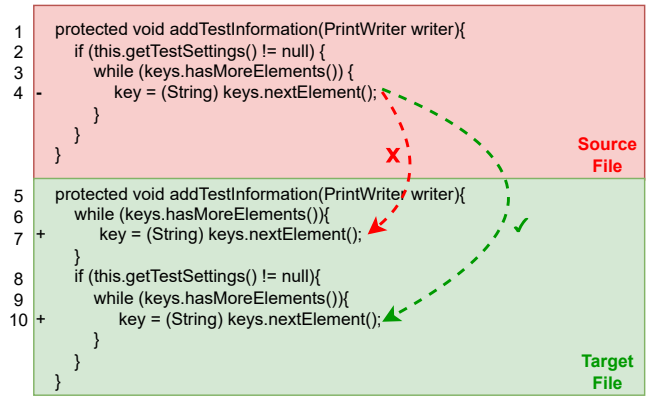

Fig. 4. Illustration of the rule ANCE using partial code changes of PerfReportGenerator.java from the commit 'e8b3bdb' of ActiveMQ.

nodes and another iterative mechanism that connects the three iterative mapping processes.

*1) **Iterative Statement Mapping**:* This step maps the statements in $\mathcal{UM}_s$ to the statements in $\mathcal{UM}_t$. For each statement $s_i \in \mathcal{UM}_s$, we collect the statements with the same label as s_i in $\mathcal{UM}_t$. Such statements are to be mapped to s_i as only the statements with the same label, e.g., `MethodDeclaration`, can be mapped. After collecting the candidates for all statements in $\mathcal{UM}_s$, we strive for mapping a statement to one of its candidates based on the similarities measured using the following five heuristic rules.

IDEN characterizes whether two statements are identical. This rule is to deal with the identical statements that have not been mapped in the identical mapping step. The IDEN between two statements is 1 if they are identical, otherwise 0.

ANCE is defined as the sum of the path lengths between two statements and their closest ancestor statements that have been mapped in the ASTs. Given two statements s_i and s_j and the pair of their closest mapped ancestor statements $\langle as_i, as_j \rangle$, we compute the path length between s_i and its ancestor statement as_i and the path length between s_j and as_j. Then, we compute ANCE as the sum of both path lengths. The smaller the ANCE is, the more likely s_i and s_j can be mapped, which can be viewed as *the principle of proximity*. In Fig. 4, there are three pairs of mapped statements: ⟨line 1, line 5⟩, ⟨line 2, line 8⟩, and ⟨line 3, line 9⟩, so that the ANCE between line 4 and line 7 is 8[2], and the ANCE between line 4 and line 10 is 2. Thus, line 4 and line 10 are more likely to be mapped.

IMSR is defined as the ratio of the identical or mapped descendant statements of two statements. For two statements s_i and s_j, we compute IMSR as $\frac{2\times|imds(s_i,s_j)|}{|ds(s_i)|+|ds(s_j)|}$, where $imds()$ obtains the set of the identical/mapped descendant statements of two statements, and $ds()$ obtains the set of descendant statements of a statement. The larger the IMSR is, the more likely s_i and s_j can be mapped. In Fig. 5, there are three method declaration statements. The IMSR between line 1 and

[2]There are three Block nodes, e.g., n_{10} in Fig. 1(b), generated for each {...} between the two statements and their closest mapped ancestor statements.

TABLE I
THE FREQUENCIES OF TEN HEURISTIC RULES APPLIED TO THE FILE REVISIONS OF TEN JAVA PROJECTS.

Rule	IDEN	ANCE	IMSR	IMTR	S-ABS	I-MSIS	I-IMTR	I-ABS	T-MSIS	T-ABS
Fequency	135,059	47,679	29,395	20,095	13,467	202,736	151,013	39,935	80,161	55,675

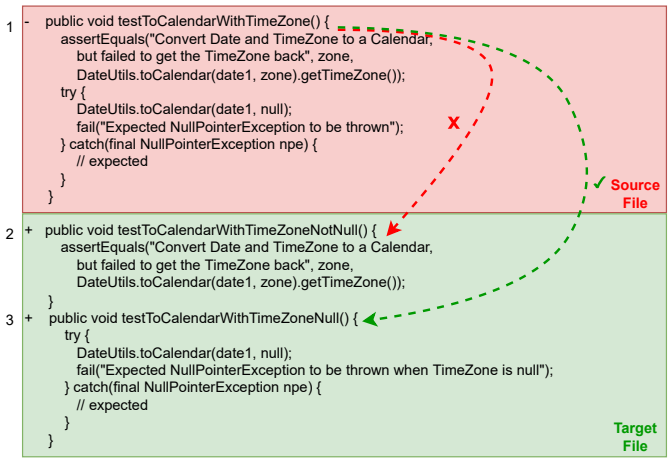

Fig. 5. Illustration of the rule IMSR using partial code changes of DateUtilsTest.java from the commit 'd9a2c69' of Commons-Lang.

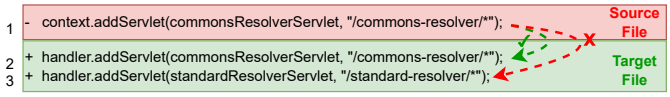

Fig. 6. Illustration of the rule IMTR using partial code changes of RequestPartIntegrationTests.java from the commit '529e629' of Spring-Framework.

line 2 is 0.33, and the IMSR between line 1 and line 3 is 0.88. Thus, line 1 and line 3 are more likely to be mapped.

IMTR is defined as the ratio of the identical or mapped descendant tokens of two statements. The statements sharing more identical/mapped tokens are more likely to be mapped. We compute IMTR between two statements s_i and s_j as $\frac{2\times|imts(s_i,s_j)|}{|ts(s_i)|+|ts(s_j)|}$, where $imts()$ obtains the set of identical/mapped descendant tokens of two statements, and $ts()$ returns the set of descendant tokens of a statement. In Fig. 6, the IMTR between line 1 and line 2 is 0.75, and the IMTR between line 1 and line 3 is 0.25. Thus, line 1 and line 2 are more likely to be mapped.

S-ABS characterizes whether the nearest above and the nearest below sibling statements of two statements are mapped. A statement often has sibling statements, e.g., the statements in the body of a method are siblings. The S-ABS between two statements can be: 1) 0 when neither the nearest above nor the nearest below siblings are mapped; 2) 1 when either the nearest above or the nearest below siblings are mapped; and 3) 2 when both the nearest above and the nearest below siblings are mapped. The larger the S-ABS is, the more likely the two statements can be mapped.

We use the five rules in the order of IDEN→ANCE→IMSR→IMTR→S-ABS based on their frequencies (i.e., the numbers of file revisions to which the rules can apply) in the file revisions from ten Java projects (see Table II), as listed in Table I. For each rule, we measure the similarities between the statements in $\mathcal{UM}_s$ and their candidates. Then, we map statement s_i to one of its candidate s_j if they achieve the *unique optimal* similarity among 1) the similarities between s_i and its candidates and 2) the similarities between s_j and the statements in $\mathcal{UM}_s$ that have s_j as a candidate. In other words, we require that two mapped statements must be the only best match among all the candidates for each other to ensure a high accuracy of mapping. After the pair of $\langle s_i, s_j\rangle$ is mapped, we add it to the global set of mapped nodes $\mathcal{M}$, remove s_j from the candidates of all statements in $\mathcal{UM}_s$, and remove s_i and s_j from $\mathcal{UM}_s$ and $\mathcal{UM}_t$, respectively. If s_i has multiple candidates with the optimal similarities, the candidate that should be mapped to s_i cannot be determined. Such candidates are further mapped to s_i using the next rule.

After going through a round of the five rules, if there are new statement mappings generated and there are still unmapped statements in $\mathcal{UM}_s$, another round of the rules is applied since the new statement mappings may provide useful information for generating more statement mappings. Specifically, the new statement mappings may affect the similarities measured using the three rules ANCE, IMSR, and S-ABS. Based on the updated similarities, it is possible that some statements can be mapped. We iteratively perform several rounds of the five rules following the same order until no new statement mappings are generated.

*2) **Iterative Inner-statement Mapping**:* This step maps the inner-statements in $\mathcal{UM}_s$ to the inner-statements in $\mathcal{UM}_t$. For each inner-statement $is_i \in \mathcal{UM}_s$, we collect the inner-statements with the same label as is_i in $\mathcal{UM}_t$. Such inner-statements are to be mapped to is_i as only two inner-statements with the same label, e.g., `MethodInvocation`, can be mapped. After collecting the candidates for all inner-statements in $\mathcal{UM}_s$, we want to map an inner-statement to one of its candidates based on the similarities measured using the following three heuristic rules.

I-MSIS characterizes whether two inner-statements belong to a pair of mapped statements or inner-statements. The I-MSIS between two inner-statements is 1 if their parent statements/inner-statements are mapped, otherwise 0.

I-IMTR is defined as the ratio of the identical or mapped descendant tokens of two inner-statements. This rule is similar to the rule IMTR used for iterative statement mapping.

I-ABS characterizes whether the nearest above and the nearest below sibling inner-statements or tokens are mapped. In a statement, the inner-statements or tokens that do not overlap are siblings. This rule is similar to the rule S-ABS used for iterative statement mapping.

We follow the order of I-MSIS→I-IMTR→I-ABS to use the three rules according to their frequencies of application listed in Table I. For each rule, we measure the similarities between the inner-statements in $\mathcal{UM}_s$ and their candidates. We map inner-statement is_i to one of its candidate is_j if their similarity is the *unique optimal* among 1) the similarities between is_i and its candidates and 2) the similarities between is_j and the inner-statements in $\mathcal{UM}_s$ that have is_j as a candidate. That is, we require that two mapped inner-statements must be the only best match among all the candidates for each other to ensure a high accuracy of mapping. After the pair of $\langle is_i, is_j \rangle$ is mapped, we add it to the global set of mapped nodes $\mathcal{M}$, remove is_j from the candidates of all inner-statements in $\mathcal{UM}_s$, and remove is_i and is_j from $\mathcal{UM}_s$ and $\mathcal{UM}_t$, respectively. If is_i has multiple candidates with the optimal similarities, we cannot determine the candidate to be mapped to is_i. Such candidates are mapped to is_i using the next rule.

After applying a round of the three rules, if there are new inner-statement mappings generated and there are still unmapped inner-statements in $\mathcal{UM}_s$, another round of the rules is applied since the new inner-statement mappings may provide useful information for generating more inner-statement mappings. Specifically, the new inner-statement mappings may affect the similarities measured using the two rules I-MSIS and I-ABS. Some inner-statements may be mapped due to the updated similarities. We iteratively perform the three rules following the same order for several rounds until there are no new inner-statement mappings generated.

*3) **Iterative Token Mapping**:* This step maps the tokens in $\mathcal{UM}_s$ to the tokens in $\mathcal{UM}_t$. For each token $t_i \in \mathcal{UM}_s$, we collect the tokens with the same label as t_i in $\mathcal{UM}_t$ as candidates to be mapped to t_i. After collecting the candidates for all tokens in $\mathcal{UM}_s$, we try to map a token to one of its candidates based on the similarities measured using the following two heuristic rules.

T-MSIS characterizes whether two tokens belong to a pair of mapped statements or inner-statements. This rule is similar to the rule I-MSIS used for iterative inner-statement mapping.

T-ABS characterizes whether the nearest above and the nearest below sibling tokens of two tokens are mapped. The tokens in a statement are siblings. This rule is similar to the rule S-ABS used for iterative statement mapping.

We apply the two rules following the order of T-MSIS→T-ABS based on their frequencies of application listed in Table I. For each rule, we measure the similarities between the tokens in $\mathcal{UM}_s$ and their candidates. We map token t_i to one of its candidate t_j if they achieve the *unique optimal* similarity among 1) the similarities between t_i and its candidates and 2) the similarities between t_j and the tokens in $\mathcal{UM}_s$ that has t_j as a candidate. That is, two mapped tokens must be the only best match among all the candidates for each other to ensure a high accuracy of mapping. After the pair of $\langle t_i, t_j \rangle$ is mapped, we add it to the global set of mapped nodes $\mathcal{M}$, remove t_j from the candidates of all tokens in $\mathcal{UM}_s$, and remove t_i and t_j from $\mathcal{UM}_s$ and $\mathcal{UM}_t$, respectively. If t_i has multiple candidates with the optimal similarities, the candidate that should be mapped to t_i cannot be determined. Such candidates are mapped to t_i using the next rule.

After performing a round of both rules, if there are new token mappings generated and there are still unmapped tokens in $\mathcal{UM}_s$, another round of the rules is applied since the new token mappings may provide useful information for generating more token mappings. Specifically, the new token mappings may affect the similarities measured using T-LRS, and thus some tokens may be further mapped. We iteratively perform the two rules following the same order for several rounds until no new token mappings are generated.

*4) **Outer Iteration Between Three Inner Iterative Mapping Processes**:* Based on our analysis, the mappings of higher-level nodes, e.g., statements (*resp.* inner-statements), have impacts on the mappings of lower-level nodes, e.g., inner-statements (*resp.* tokens), and vice versa, which are reflected by the heuristic rules IMTR, I-MSIS, I-IMTR, I-ABS, and T-MSIS. As shown in Fig. 2, we build an outer iteration between the three inner iterative mapping processes by connecting the iterative token mapping step to the iterative statement mapping step. After an iterative token mapping process is finished, if there are new token mappings generated, we go back to the iterative statement mapping step.

D. Code Edit Action Generation

We generate a series of code edit actions from the three types of mapped and unmapped nodes to describe the code changes in the file revision. Our code edit actions describe code changes in terms of source code elements (i.e., statements, inner-statements, and tokens) and can be understood by users without tracking the nodes in the source and target ASTs. Specifically, we define seven types of code edit actions:

- $del(e)$ deletes the element e.
- $add(e)$ adds an element e.
- $upd(e_s, e_t)$ replaces the element e_s with an element e_t.
- $mov(e_s, e_t)$ moves the element e_s to the position of the element e_t. e_s and e_t have different parents in the ASTs.
- $exc(e_s, e_t)$ moves the element e_s to the position of the element e_t. e_s and e_t have the same parent but their relative orders are different in the ASTs.
- $m\&u(e_s, e_t)$ combines $mov(e_s, e_t)$ and $upd(e_s, e_t)$.
- $e\&u(e_s, e_t)$ combines $exc(e_s, e_t)$ and $upd(e_s, e_t)$.

To generat the code edit actions, we extend the set of pairs of mapped elements, $\mathcal{M}$. For each unmapped element $e_i \in \mathcal{UM}_s$, we add $\langle e_i, null \rangle$ to $\mathcal{M}$. For each unmapped element $e_j \in \mathcal{UM}_t$, we add $\langle null, e_j \rangle$ to $\mathcal{M}$. For each pair of $\langle e_i, e_j \rangle \in \mathcal{M}$, we generate code edit actions as follows.

- $del(e_i)$ if e_j is $null$.
- $add(e_j)$: if e_i is $null$.
- $upd(e_i, e_j)$ if e_i and e_j are not identical.
- $mov(e_i, e_j)$ if the parents of e_i and e_j are not mapped.
- $exc(e_i, e_j)$ if the parents of e_i and e_j are mapped but their relative orders in the parents are changed.
- $m\&u(e_t, e_j)$ if both $upd(e_i, e_j)$ and $mov(e_i, e_j)$ are generated from $\langle e_i, e_j \rangle$, then we replace the two actions with a compound action $m\&u(e_i, e_j)$.

TABLE II
DESCRIPTIVE STATISTICS OF TEN JAVA PROJECTS.

Project	#Commits	#File Revisions	#Code Lines	#Changed Code Lines
ActiveMQ	8,066	24,821	21,531,689	708,309(3.29%)
Commons-IO	1,067	2,726	3,010,133	108,763(3.61%)
Commons-Lang	3,032	6,915	14,866,513	323,419(2.18%)
Commons-Math	4,239	18,120	15,206,867	622,995(4.10%)
Hibernate-ORM	10,158	51,642	41,001,030	1,584,700(3.87%)
Hibernate-Search	5,368	26,865	11,362,103	495,213(4.36%)
Junit4	1,241	3,802	29,076,733	1,211,965(9.89%)
Netty	11,138	39,778	55,596,221	2,396,919(4.17%)
Spring-Framework	4,112	16,434	10,199,157	546,461(5.36%)
Spring-Roo	4,274	19,894	12,169,842	1,350,164(11.09%)
Total	**52,695**	**210,997**	**159,620,332**	**7,070,286(4.43%)**

- $e\&u(e_i, e_j)$ if both $upd(e_i, e_j)$ and $exc(e_i, e_j)$ are generated from $\langle e_i, e_j \rangle$, then we replace the two actions with a compound action $e\&u(e_i, e_j)$.

Given the partial code changes shown in Fig. 1(a), iASTMapper finally generates nine code edit actions related to four statements, two inner-statements, and three tokens, as shown in the right below part of Fig. 1(c). For example, the statement edit action *upd*(`MethodDeclaration`(Line:1), `MethodDeclaration`(Line:1)) implies that the method declaration statement in line 1 of the source file is updated to the method declaration statement in line 1 of the target file. To factilitate understanding of the actions, we present the line numbers and indexes of the elements in each action.

IV. EVALUATION

In this section, we evaluate iASTMapper by answering three research questions. After describing the experimental dataset, we present the answer to each research question. Our experimental environment is a desktop PC with Intel Core i5-10500 CPU, 16G RAM, and Windows 10 Operating System.

We download ten open-source Java projects from GitHub. These projects are analyzed in prior studies [4], [5]. For each project, we collect the commits and the file revisions involved in the commits. For each file revision, we count the lines of code (LOC) of the two source code files and also count the code churns, i.e., changed code lines, in both files. Then, we count all code lines and all changed code lines in the file revisions of a project and compute the percentage of changed code lines. Table II lists the statistics of the ten projects. In total, there are 52,695 commits and 210,997 file revisions.

A. *RQ1: How effective is iASTMapper compared with state-of-the-art AST mapping algorithms?*

Motivation. We want to quantitatively evaluate the performance of iASTMapper by comparing it with state-of-the-art AST mapping algorithms on a large-scale dataset of file revisions.

Approach. We select three AST mapping algorithms widely used in prior studies as our baselines, namely

- **GumTree (GT)** [2] uses two phases to map the nodes between the source and target ASTs. In the first phase, a greedy top-down algorithm is used to map identical subtrees. In the second phase, a bottom-up algorithm is used to map the nodes that share a significant number of mapped descendants. After that, GT tries to map the unmapped descendants of those nodes.
- **MTDiff (MTD)** [3] first removes unchanged subtrees from the ASTs using an identical subtree optimization. Then, MTD maps AST nodes using the ChangeDistiller algorithm [12]. Another four optimizations are used to find additional node mappings.
- **IJM** [4] is an AST mapping algorithm for Java programs. IJM first reduces the ASTs by pruning many name nodes[3] and merging the value of each pruned node to its parent node. Then, IJM splits the ASTs into parts along each declaration and maps AST nodes from the parts using an adaptation of the GT algorithm. The adaptation evaluates the similarity of node values when mapping two nodes.

For each file revision of the ten projects, we use the Eclipse JDT parser to generate ASTs for the source file and target file. Then, we use GT, MTD, and iASTMapper to map nodes between the two ASTs, respectively. For IJM, the ASTs are reduced using the AST reduction algorithm proposed in IJM. Then, we use IJM to map nodes between the reduced ASTs. To compare iASTMapper and IJM, we also apply iASTMapper to the reduced ASTs. We refer to the modified iASTMapper that uses the reduced ASTs as **iASTMapper(IJM-AST)**. Note that all these algorithms fail to analyze 134 file revisions because of errors happened when extracting information from the revisions using Git operations. Using the AST node mappings generated by each algorithm, we produce a series of AST edit actions using Chawathe et al.'s algorithm [12] and also produce a series of code edit actions using our algorithm described in Section III-D.

We automatically evaluate the algorithms using three metrics: 1) #**AST Node Mappings**: measures the number of AST node mappings, which can reflect the mapping capability of an algorithm to a certain degree. The larger number of mapped AST nodes is, the higher possibility that the algorithm can generate more accurate AST edit actions; 2) **AST Edit Script (ES) Size**: calculates the number of AST edit actions, which is used as an indicator of the cognitive efforts required by users to understand the code changes. The larger size of the AST edit script is, the more efforts are required; and 3) **Code Edit Script (ES) Size**: computes the number of code edit actions. As explained in Section III-D, code edit actions are proposed in this work to overcome the drawback of AST edit actions. A larger code edit script also requires more efforts to understand. #AST Node Mappings and AST ES Size are widely used in the automatic evaluations of AST mapping algorithms [2]–[4], while Code ES Size is proposed in this work.

For an algorithm, A, we measure the value of a metric, m, for each file revision, r, which is denoted as $m_r(A)$. Next, we measure the increased or decreased degree of iASTMapper over each of the other four algorithms, e.g., B, in terms of m

[3]Name nodes, e.g., the node n_5 in Fig. 1(b), are children of various other nodes like method or type declarations. A name node contains the name (e.g., a method or type name) of its parent node as its value.

TABLE III
THE TOTAL RESULTS GENERATED FOR THE FILE REVISIONS OF TEN JAVA PROJECTS USING FIVE ALGORITHMS. THE PERCENTAGE IN A CELL REPRESENTS THE INCREASED ('+') OR DECREASED ('-') DEGREE OF IASTMAPPER OVER THE CORRESPONDING ALGORITHM.

	iASTMapper	**iASTMapper(IJM-AST)**	**GT**	**MTD**	**IJM**
#AST Node Mappings	495,686,699	390,840,846(**+26.83%**)	395,359,204(**+25.38%**)	395,975,927(**+25.18%**)	376,977,471(**+31.49%**)
AST ES Size	14,281,762	9,528,528(**+49.88%**)	14,335,399(**-0.37%**)	16,775,546(**-14.87%**)	10,222,981(**+39.70%**)
Code ES Size	2,627,196	2,652,314(**-0.95%**)	2,661,412(**-1.29%**)	2,807,658(**-6.43%**)	2,798,826(**-6.13%**)

TABLE IV
STATISTICS OF THREE METRICS ACHIEVED BY FIVE ALGORITHMS. 'P' STANDS FOR 'PERCENTILE'.

		Min	**Max**	**Mean**	**Std**	**P25**	**P50**	**P75**	**P95**
#AST Node Mappings	**GT**	1	54,095	1,874.96	3,150.75	342.00	821.00	2,022.00	7,177.00
	MTD	7	54,095	1,877.88	3,152.29	344.00	823.00	2,026.00	7,182.00
	IJM	7	53,395	1,787.78	3,040.26	318.00	772.00	1,920.00	6,867.00
	iASTMapper(IJM-AST)	4	55,761	1,853.53	3,160.13	330.00	801.00	1,989.00	7,106.00
	iASTMapper	4	66,996	2,350.75	4,031.15	411.00	1,008.00	2,528.00	9,093.00
AST ES Size	**GT**	0	22,816	67.98	239.06	4.00	14.00	51.00	279.00
	MTD	0	22,880	79.56	259.75	4.00	16.00	61.00	335.00
	IJM	0	15,659	48.48	172.98	3.00	10.00	37.00	196.00
	iASTMapper(IJM-AST)	0	15,638	45.19	154.08	3.00	10.00	35.00	183.00
	iASTMapper	0	22,788	67.73	234.43	4.00	14.00	51.00	279.00
Code ES Size	**GT**	0	2,380	12.62	35.61	2.00	4.00	11.00	49.00
	MTD	0	2,351	13.32	36.51	2.00	4.00	12.00	52.00
	IJM	0	2,455	13.27	39.08	2.00	4.00	11.00	52.00
	iASTMapper(IJM-AST)	0	2,372	12.58	33.96	2.00	4.00	11.00	49.00
	iASTMapper	0	2,382	12.46	34.02	2.00	4.00	11.00	48.00

as $\frac{\sum_{r\in R} m_r(iASTMapper)-\sum_{r\in R} m_r(B)}{\sum_{r\in R} m_r(B)}$, where R denotes the entire set of file revisions.

Results. iASTMapper outperforms state-of-the-art AST mapping algorithms, i.e., GT, MTD, and IJM, in terms of #AST Node Mappings and Code ES Size. Table III presents the automatic evaluation results. iASTMapper generates the maximum number of AST node mappings, which increases 25.38%, 25.18%, and 31.49% by comparing with GT, MTD, and IJM, respectively. This result shows that iASTMapper has a more powerful capability to map AST nodes. Moreover, iASTMapper generates the minimum size of code edit scripts, which decreases the size by 1.29%, 6.43%, and 6.13% in comparison with GT, MTD, and IJM, respectively. The shorter code edit scripts generated by iASTMapper for file revisions can reduce the burden for users to understand code changes.

In terms of AST ES size, iASTMapper is better than MTD, very close to GT, but worse than IJM. The worse result is caused by the fact that IJM reduces the standard ASTs by removing many name nodes and thus does not generate mappings of those nodes, which contributes to smaller sizes of AST edit scripts. However, the modified iASTMapper that uses the reduced ASTs, iASTMapper(IJM-AST), generates smaller sizes of AST edit scripts than IJM, which shows the superiority of iASTMapper over IJM. Although iASTMapper(IJM-AST) generates smaller sizes of AST edit scripts than iASTMapper, iASTMapper generates smaller sizes of code edit scripts than iASTMapper(IJM-AST). This result indicates that the reduced ASTs has negative impacts on the generation of code edit scripts, for the following reason. Some of the removed name nodes may be necessary to generate concise code edit actions, e.g., the compound actions $m\&u$ and $e\&u$ (see Section III-D); and removing such nodes lead to more code edit actions.

Table IV further presents several statistics of the three metric values obtained by the five algorithms for all file revisions. Note that for each algorithm, both the minimum AST ES Size and the minimum Code ES Size are 0, while the minimum #AST node mappings is not 0. This discrepancy arises from some file revisions where the source file and target file possess identical content. An example can be found in the case of the BrokerService.java of the commit 'ba6e62c' from ActiveMQ. Moreover, we observe two prevalent trends in the metric values produced by the algorithms, namely: 1) The metric values exhibit a large range from 0 or below ten to thousands or tens of thousands; and 2) The majority of metric values fall within a confined range, e.g., the code ES sizes generated by iASTMapper for 95% of the revisions are no more than 48.

*B. **RQ2:** Is it important to apply the iterative mechanisms in iASTMapper?*

Motivation. In iASTMapper, we apply two types of iterative mechanisms, i.e., the three inner iterative mechanisms used for mapping each of the three types of nodes and the outer iterative mechanism that connects the inner iterative mapping mechanisms. We want to investigate the importance of the inner and outer iterative mechanisms to iASTMapper.

Approach. To separately study the importance of the inner and outer iterative mechanisms, we implement two simplified variants of iASTMapper: 1) **iASTMapper-Inner** which removes the three inner iterative mechanisms and applies one round of the heuristic rules (e.g., IDEN) to map each of the three types of nodes; and 2) **iASTMapper-Outer** which removes the outer iterative mechanism by disconnecting the iterative token mapping step from the iterative statement mapping step.

We randomly sample 50 file revisions. For each file revision, we use iASTMapper and both variants to generate the AST node mappings. Then, we generate AST edit actions and

TABLE V
COMPARISON OF IASTMAPPER AND ITS TWO VARIANTS.

	#AST Node Mappings	AST ES Size	Code ES Size	#Correct Code Edit Actions (Accuracy)
iASTMapper	114,700	1,451	337	332(**98.52%**)
iASTMapper-Inner	112,145	6,325	1,089	280(**25.71%**)
iASTMapper-Outer	113,589	4,691	345	324(**93.91%**)

code edit actions from the mappings. The first two co-authors who have five and 11 years of Java programming experience, respectively, manually evaluate the correctness of 1,165 code edit actions of statements generated for the file revisions by the three algorithms. Note that we do not evaluate the code edit actions of inner-statements and tokens for two reasons: 1) Statements are the basic building blocks for programming. The changes of statements provide more concise summary of the code changes; and 2) There are 1,740 code edit actions of inner-statements and 4,577 code edit actions of tokens, which could significantly increase the evaluation efforts. Both co-authors independently evaluate the correctness of each of the 1,165 actions. There are 63 actions with disagreement. We measure the inter-rater agreement using the Fleiss Kappa [35], and the Kappa value is 0.85, indicating almost perfect agreement. After discussing the actions with disagreement, both co-authors reach a consensus on the correctness of all actions. The **accuracy** of each algorithm is measured as the percentage of the correctly generated actions.

Results. The inner and outer iterative mechanisms are both important to iASTMapper. Moreover, the inner iterative mechanisms contribute more to iASTMapper than the outer iterative mechanism. Table V presents the evaluation results of iASTMapper and its two variants. iASTMapper outperforms both variants in terms of the three automatic evaluation metrics (defined in RQ1) and the accuracy of code edit actions. iASTMapper-Inner is worse than iASTMapper-Outer. Specifically, iASTMapper generates 114,700 AST node mappings, which slightly increase 2.28% and 0.98% of AST node mappings from iASTMapper-Inner and iASTMapper-Outer, respectively. iASTMapper generates 1,451 AST edit actions and decreases 77.06% and 69.07% of AST edit actions from iASTMapper-Inner and iASTMapper-Outer, respectively. In addition, iASTMapper generates 337 code edit actions and decreases 69.05% and 2.32% of code edit actions from iASTMapper-Inner and iASTMapper-Outer, respectively.

iASTMapper achieves the highest accuracy, 98.52%, while iASTMapper-Outer and iASTMapper-Inner have 93.91% and 25.71% accuracy, respectively. This result shows that the inner iterative mechanisms contribute more to the accuracy of iASTMapper than the outer iterative mechanism. Moreover, the result confirms our observations that 1) Multiple rounds of applying the heuristic rules are important to generate stable and accurate mappings of the nodes in a specific level; and 2) There is mutual influence between the mappings of nodes in different levels. Connecting the inner iterative mapping processes helps generate better node mappings.

C. *RQ3: Can iASTMapper generate more accurate AST mappings than state-of-the-art algorithms?*

Motivation. In RQ1, we have verified the superiority of iASTMapper to the three state-of-the-art baselines in producing a larger number of AST node mappings and shorter sizes of AST/code edit scripts for file revisions. In this research question, we want to investigate another important question, i.e., whether the AST node mappings and the AST/code edit actions produced by iASTMapper are more accurate than those produced by the baselines.

Approach. We manually evaluate the accuracy of the code edit actions generated by iASTMapper, GT, MTD, and IJM. We choose to evaluate the code edit actions rather than the AST node mappings or AST edit actions because: 1) AST/Code edit actions are generated from the AST node mappings. The accuracy of AST/code edit actions can reflect the accuracy of AST node mappings; and 2) As code edit actions are presented using the program elements that users are familiar with, they can facilitate the manual evaluation.

We randomly select 200 file revisions and randomly divide them into four groups. Each group has 50 file revisions. We also recruit 12 evaluators (including 11 masters and one undergraduate) who are interested in our study and willing to perform the manual evaluation task from the first co-author's university. The evaluators have 2-5 years of experience in Java programming. We divide the evaluators into four groups with each group having three members, while ensuring that the members in different groups have comparative Java programming experience. Next, we randomly assign the four file revision groups to the four evaluator groups.

After introducing the evaluation task to the evaluators with an example, we ask them to evaluate the correctness of the four sets of code edit actions of statements generated for their allocated file revisions using GT, MTD, IJM, and iASTMapper. In total, there are 1,455, 1,489, 1,572, and 1,580 code edit actions of statements generated for the 50 file revisions in the four groups, respectively. The evaluators do not know which algorithm is used to generate each set of code edit actions. They first independently evaluate the correctness of each action, and label the correct actions with 1; otherwise 0. There are 47, 119, 89, and 100 actions with disagreement among the evaluators in the four groups, respectively. We measure the inter-rater agreement among the evaluators in each group, and the Fleiss Kappa values are 0.82, 0.81, 0.84, and 0.84, respectively, indicating almost perfect agreement. The evaluators in each group discuss the actions with disagreement and reach a consensus on the correctness of all actions.

For each algorithm, we count the code edit actions of statements generated for the 200 file revisions and the actions evaluated as correct. We also distinguish the types of the actions (see Section III-D). Then, the accuracy of an algorithm is calculated as the percentage of correct actions, with respect to the entire set of actions or the actions of every specific type.

Results. iASTMapper generates 96.23% accurate code edit actions and improves GT, IJM, and MTD by

TABLE VI
THE EVALUATION RESULTS OF CODE EDIT ACTIONS GENERATED FOR 200 FILE REVISIONS USING FOUR ALGORITHMS. THE VALUE '$n/m(p)$' IN A CELL MEANS THAT n OF THE m ACTIONS GENERATED BY THE CORRESPONDING ALGORITHM ARE EVALUATED AS CORRECT AND THE ACCURACY IS p.

	ADD	DEL	UPD	MOV	EXC	M&U	E&U	Total
GT	562/596(94.30%)	170/173(98.27%)	492/527(93.36%)	45/87(51.72%)	6/7(85.71%)	14/15(93.33%)	4/7(57.14%)	1293/1412(91.57%)
MTD	460/574(80.14%)	139/151(92.05%)	494/649(76.12%)	54/117(46.15%)	18/36(50.00%)	33/111(29.73%)	5/42(11.90%)	1203/1680(71.61%)
IJM	623/666(93.54%)	222/243(91.36%)	480/554(86.64%)	50/82(60.98%)	6/7(85.71%)	15/16(93.75%)	3/5(60.00%)	1399/1573(88.94%)
iASTMapper	611/639(**95.62%**)	215/216(**99.54%**)	454/476(**95.38%**)	75/77(**97.40%**)	2/2(**100.00%**)	16/17(**94.12%**)	4/4(**100.00%**)	1377/1431(**96.23%**)

TABLE VII
AVERAGE RUNNING TIMES (MILLISECONDS) OF FOUR ALGORITHMS.

Project	**GT**	**MTD**	**IJM**	**iASTMapper**
activemq	12.49	187.46	40.78	47.78
commons-io	17.26	430.61	33.98	59.80
commons-lang	35.56	1,115.06	58.63	151.45
commons-math	18.31	447.41	47.77	156.83
hibernate-orm	12.49	207.50	52.81	49.36
hibernate-search	7.19	167.74	33.35	72.31
junit4	6.05	49.23	24.25	15.78
netty	14.15	284.79	40.25	341.36
spring-framework	10.67	186.07	38.39	36.56
spring-roo	14.61	273.55	76.57	66.56
Overall	**13.38**	**269.44**	**46.66**	**119.86**

TABLE VIII
STATISTICS OF THE AVERAGE RUNNING TIMES (MILLISECONDS) OF FOUR ALGORITHMS. 'P' STANDS FOR 'PERCENTILE'.

	Min	Max	Mean	Std	P25	P50	P75	P95
GT	0.0	11,815.2	13.38	65.27	3.00	5.00	11.20	42.40
MTD	0.0	171,684.4	269.44	2,323.24	4.60	15.40	75.40	807.76
IJM	0.0	34,271.2	46.66	232.43	4.00	9.20	25.60	166.20
iASTMapper	0.0	436,209.0	119.86	4,166.16	6.20	14.20	39.00	192.76

4.66%, 7.29%, and 24.62%, respectively. The accuracy of iASTMapper is 94.12%-100% with respect to the different types of actions, which is more stable than the baselines. Table VI presents the manual evaluation results. 1,412, 1,680, 1,573, and 1,431 code edit actions of statements are generated for the 200 file revisions by GT, MTD, IJM, and iASTMapper, respectively. This result is almost consistent with the result listed in Table III. However, the number of actions generated by iASTMapper is slightly greater than those generated by GT. This result may be caused by the reason that the distribution of the sampled dataset is not very close to the entire dataset.

The accuracy of the code edit actions generated by GT, MTD, IJM, and iASTMapper are 91.57%, 71.61%, 88.94%, and 96.23%, respectively. Therefore, the performance ranking of these algorithms is iASTMapper $>$ GT $>$ IJM $>$ MTD. iASTMapper achieves an improvement of 4.66%, 7.29%, and 24.62% in comparison with GT, IJM, and MTD, respectively. In terms of the seven types of actions, iASTMapper also achieves the highest accuracy, ranging from 94.12% to 100%. Unlike iASTMapper, the accuracy of the baselines has a large fluctation, with respect to the different types of actions. The fluctation ranges of GT, MTD, and IJM are 51.72%-98.27%, 11.9%-92.05%, and 60%-93.75%, respectively.

*D. **RQ4:** How efficient is iASTMapper?*

Motivation. In RQ1 and RQ3, we have demonstrated the effectiveness of iASTMapper. However, if iASTMapper cannot produce the analysis result for a file revision in a reasonable time, it may not be acceptable in practice. In this research question, we investigate the efficiency of iASTMapper by comparing it with state-of-the-art AST mapping algorithms.

Method. We measure the running time (in milliseconds) used to generate ASTs, build AST node mappings, and produce code edit actions by GT, MTD, IJM, and iASTMapper for the file revisions from ten projects. For each algorithm, we perform it on a file revision for five times and calculate the average running time. We then compute the average of the average running times over the file revisions within a project and the overall average running time over all file revisions.

Results. On average, iASTMapper takes approximately 119.86 milliseconds (ms) to locate the code changes in a file revision. Table VII presents the average running times of four algorithms. In terms of the overall average running time, the order of the algorithms is GT $>$ IJM $>$ iASTMapper $>$ MTD. iASTMapper requires approximately 119.86ms to process a file revision from building ASTs to generating code edit actions, which is better than MTD (i.e., 269.44ms) but worse than GT (i.e., 13.38ms) and IJM (i.e., 46.66ms). We also observe that compared with GT and IJM, the average running time of iASTMapper and MTD on different projects have a relatively large fluctation. The possible reason is that the mapping algorithms used by iASTMapper and MTD are more complex than those used by GT and IJM, and thus it takes more time for iASTMapper and MTD to analyze the file revisions with a lot of AST nodes from some projects, such as Commons-Lang, Commons-Math, and Netty.

To better understand the efficiency of the four algorithms, we further calculate several statistics of the average running times spent by each algorithm on all file revisions, as listed in Table VIII. The average running times of each algorithm exhibit a large range, spanning from 0ms (which indeed means "$<$1ms") to tens of thousands or hundreds of thousands of milliseconds. However, we observe that iASTMapper requires 14.2ms for 50% of the file revisions and 192.76ms for 95% of the revisions, indicating that iASTMapper can complete the analysis of most file revisions within 0.2 seconds.

V. THREATS TO VALIDITY

Threats to internal validity relate to the errors in the implementation of the algorithms, the competency of the evaluators for evaluating code edit actions, and the evaluators' bias in the evaluation. We implement the three baselines: GumTree, MTDiff, and IJM, using their packages released at GitHub, and carefully check the code of our iASTMapper

and its variants to ensure that there are no errors with the implementation. For the manual evaluation task, we recruit 12 evaluators with 2-5 years of Java programming experience. Before conducting the task, we use an example to explain the task to the evaluators and make sure that the evaluators understand the task and thus reduce possible mistakes during the evaluation. To avoid the subjective bias of a single person, the evaluation of each code edit action is performed by two or three evaluators. The evaluators first independently evaluate a set of code edit actions and then discuss the disagreements together to reach a consensus.

Threats to external validity relate to the generalizability of the results. Our automatic evaluation is conducted using a large-scale dataset consisting of 210,997 file revisions from ten Java projects. The results obtained from such a big dataset should be reliable. Moreover, we conduct a manual analysis to evaluate the accuracy of the code edit actions generated by iASTMapper and the baselines. Due to the heavy workload of the manual evaluation, we randomly sample 50 file revisions and 200 file revisions for answering the research questions RQ2 and RQ3, respectively. Two co-authors and 12 evaluators evaluate the actions generated for the two sets of file revisions, respectively. The sample sizes and the numbers of evaluators are not very large, but they are similar to or larger than those used in prior studies [2]–[5]. The results could provide an insight of the good performance of iASTMapper.

VI. Related work

A. AST mapping algorithms

There are many AST mapping algorithms proposed in prior studies [7]–[10]. For example, ChangeDistiller [12] is proposed by Fluri et al. to calculate the mappings of AST nodes based on a reduced AST. Pawlik et al. [11] propose an efficient method, RTED, to calculate the AST edit actions from the AST node mappings. In recent years, GumTree [2] uses a top-down algorithm to map identical subtrees and a followed bottom-up algorithm to map the nodes that share a significant number of mapped descendants. MTDiff uses an identical subtree optimization to prune ASTs by removing unchanged subtrees and then uses another four optimizations to find additional mappings of nodes. IJM [4] improves the accuracy of AST edit actions by pruning the ASTs and splitting the reduced ASTs into parts with similar values to determine whether the nodes can be mapped or not. Fan et al. [5] propose a differential testing approach that can automatically detect inaccurate mappings generated by multiple AST mapping algorithms. They investigate the three widely used algorithms: GumTree, MTDiff, and IJM, and find that those algorithms generate inaccurate mappings for at least 20% file revisions from ten Java projects. Inspired by the study, we propose a novel iterative similarity-based AST mapping algorithm to generate more accurate AST mappings. Moreover, we propose a method for generating code edit actions (a novel concept proposed in this work) from the AST node mappings to help users better understand the code changes.

B. Downstream tasks of AST mapping

AST mapping is the underlying basis of many downstream tasks, such as identifying non-essential changes [27] and locating repetitive edits [28]. Besides, GumTree is widely used to analyze change patterns, such as bug-fixing changes [30]–[32], logging changes [33] and changes to online code examples [25]. Zhang et al. [26] propose an automated adaptation analysis technique based on GumTree. Seader [29] compares edit actions generated from code snippets to prepare for data-dependency analysis. FIRA [34] also uses the AST edit actions for commit message generation. With the ability to generate more accurate AST mappings, our iASTMapper can facilitate these downstream tasks.

VII. Conclusion and Future Work

In this paper, we propose an iterative similarity-based AST mapping algorithm, iASTMapper. Our algorithm aims to generate accurate node mappings between two ASTs while sacrificing some efficiency. iASTMapper is based on three types of AST nodes in different levels: statements, inner-statements, and tokens. We first reduce the mapping problem by matching unchanged statements and inner-statements and then use an iterative method to map the remaining unmapped nodes. We iteratively map each of the three types of nodes based on their similarities measured using heuristic rules. Moreover, we iteratively connect the three iterative mapping processes. Finally, a series of code edit actions are generated from the node mappings to help users understand the code changes. We compare the performance of iASTMapper and three baselines, i.e., GumTree, MTDiff, and IJM, using 210,997 file revisions from ten Java projects and also perform a manual evaluation with 12 evaluators on 200 file revisions. The results demonstrate that iASTMapper outperforms the baselines by increasing at least 25.18% AST node mappings and improves at least 4.66% accuracy of the code edit actions. In future work, we plan to improve iASTMapper by considering more heuristic rules to address more complicate code changes, e.g., code refactoring. We will further evaluate iASTMapper with more evaluators on more file revisions.

VIII. Data Availability

We release the code and data of this work at GitHub [1] to help researchers reproduce and extend our study. In the repository, we also provide instructions on how to run our proposed iASTMapper and the baselines used in our study.

Acknowledgment

This work is supported by the Guangdong Basic and Applied Basic Research Foundation (2023A1515012292) and the National Natural Science Foundation of China (62032025).

References

[1] https://github.com/anonym00001/iASTMapper

[2] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In 29th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 313-324, 2014.

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

[4] V. Frick, T. Grassauer, F. Beck, and M. Pinzger. Generating accurate and compact edit scripts using tree differencing. In 34th IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 264-274, 2018.

[5] 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 43rd IEEE/ACM International Conference on Software Engineering (ICSE), pages 1174-1185, 2021.

[6] P. Bille. A survey on tree edit distance and related problems. Theoretical Computer Science, 337(1-3):217-239, 2005.

[7] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. ACM Transactions on Algorithms (TALG), 6(1), 1-19, 2009.

[8] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In European Symposium on Algorithms, pages 91-102, 1998.

[9] K. C. Tai. The tree-to-tree correction problem. Journal of the ACM (JACM), 26(3), 422-433, 1979.

[10] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. SIAM journal on computing, 18(6):1245-1262, 1989.

[11] M. Pawlik and N. Augsten. RTED: A robust algorithm for the tree edit distance. PVLDB, 5(4):334-345, 2011

[12] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In ACM Sigmod Record (SIGMOD), pages 493-504, 1996.

[13] G. Cobena, A. Serge, and M. Amelie. Detecting changes in XML documents. In 18th International Conference on Data Engineering(ICDE), 2002.

[14] A. Duley, C. Spandikow, and M. Kim. Vdiff: A program differencing algorithm for verilog hardware description language. In 27th IEEE/ACM International Conference on Automated Software Engineering (ASE), 19(4):459-490, 2012.

[15] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. In IEEE Transactions on Software Engineering (TSE), 38(5):1008-1026, 2012.

[16] M. Hashimoto and A. Mori. Diff/TS: A tool for fine-grained structural change analysis. In 15th Working Conference on Reverse Engineering (WCRE), pages 279-288, 2008.

[17] M. Hashimoto, A. Mori, T. Izumida. A comprehensive and scalable method for analyzing fine-grained source code change patterns. In 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 351-360, 2015.

[18] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In 19th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 2-13, 2004.

[19] Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In 20th IEEE/ACM International Conference on Automated software engineering (ASE), pages 54-65, 2005.

[20] M. J. Decker. srcdiff: Syntactic differencing to support software maintenance and evolution. Kent State University, 2017.

[21] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In 20th IEEE International Conference on Software Maintenance (ICSM), pages 188-197, 2004.

[22] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. In 11st IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MODELS), pages 326-340, 2008.

[23] J. Matsumoto , Y. Higo, S. Kusumoto. Beyond gumtree: A hybrid approach to generate edit scripts. In 16th IEEE/ACM International Conference on Mining Software Repositories (MSR), pages 550-554, 2019.

[24] V. Frick. Understanding software changes: Extracting, classifying, and presenting fine-grained source code changes. In 42nd IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pages 226-229, 2020.

[25] A. Fujimoto, Y. Higo, J. Matsumoto, and S. Kusumoto. Staged Tree Matching for Detecting Code Move across Files. In 28th IEEE/ACM International Conference on Program Comprehension (ICPC), pages 396-400, 2020.

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

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

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

[29] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. Yao, and N. Meng. Example-Based Vulnerability Detection and Repair in Java Code. In 30th IEEE/ACM International Conference on Program Comprehension (ICPC), pages 190-201, 2022.

[30] 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

[31] A. Koyuncu, K. Liu, T. F. Bissyande, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. In Empirical Software Engineering(ESE), pages 1-45, 2020

[32] 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.

[33] S. Li, X. Niu, Z. Jia, J. Wang, H. He, and T. Wang. Logtracker: Learning log revision behaviors proactively from software evolution history. In 26th IEEE/ACM International ConferenceConference on Program Comprehension (ICPC), pages 178-188, 2018.

[34] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao. FIRA: Fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation, In 44th IEEE/ACM International Conference on Software Engineering (ICSE), pages 970-981, 2022.

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

[36] 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.