Generating Accurate and Compact Edit Scripts using Tree Differencing

Veit Frick*, Thomas Grassauer*, Fabian Beck†, Martin Pinzger*
*Alpen-Adria-Universität Klagenfurt †University of Duisburg-Essen
Email: {veit.frick, thomas.grassauer, martin.pinzger}@aau.at, fabian.beck@paluno.uni-due.de

***Abstract*—For analyzing changes in source code, edit scripts are used to describe the differences between two versions of a file. These scripts consist of a list of actions that, applied to the source file, result in the new version of the file. In contrast to line-based source code differencing, tree-based approaches such as GumTree, MTDIFF, or ChangeDistiller extract changes by comparing the abstract syntax trees (AST) of two versions of a source file. One benefit of tree-based approaches is their ability to capture moved (sub)trees in the AST. Our approach, the Iterative Java Matcher (IJM), builds upon GumTree and aims at generating more accurate and compact edit scripts that capture the developer's intent. This is achieved by improving the quality of the generated move and update actions, which are the main source of inaccurate actions generated by previous approaches. To evaluate our approach, we conducted a study with 11 external experts analyzed the accuracy of 2400 randomly selected edit actions. Comparing IJM to GumTree and MTDIFF, the results show that IJM provides better accuracy for move and update actions and is more beneficial to understanding the changes.**

***Index Terms*—change extraction, tree differencing, abstract syntax trees, software evolution**

I. Introduction

Edit scripts describe the differences between two versions of a source code file. Such scripts consist of a list of actions that, when applied to a given source code file, correctly transfers it from one version of that file to another. Version control systems, such as git [1], often provide tools like diff. diff and its variations are based on Myers differencing algorithm [2] and compute edit scripts on a line-based granularity. This allows the users to view which lines have changed by presenting inserted and deleted lines. While such edit scripts are quickly generated and provide an overview, they suffer from two problems: Firstly, they are restricted to insert and delete actions and do not take updated or moved code into account. Secondly, their line-based structure is coarse grained and does not consider syntax information.

To solve these issues, approaches such as ChangeDistiller [3], GumTree (GT) [4], and MTDIFF (MTD) [5] compare the ASTs parsed from the source code files instead of their textual representation. This allows the algorithms to refine the granularity down to the level of single nodes in the AST. A small change, like an added parameter in a method call, does now show as an inserted node in the AST and not as a new line of text. Additionally, moved subtrees are taken into account, for instance to detect a change in statement order.

Such fine grained edit scripts can be used as foundation for higher level applications or further research. For example, GT or an adaption of it is used in [6] for API code recommendation, in [7] for analyzing changes in Maven build files, in [8] for automated program repair, and in [9] for discovering bug patterns in Javascript. ChangeDistiller is used to detect non-essential modifications in source-code [10] and to automate repetitive edits [11]. For these approaches to work best, it is important to have accurate edit scripts.

Existing state-of-the-art approaches, namely GT and MTD, generate edit scripts that are correct in the sense of transforming one AST into another. However, in a manual investigation, we found that those edit scripts can consist of actions, especially *move* and *update* actions, that can be classified as inaccurate. Consider an edit script where every node of the original AST is deleted and every node of the new AST is inserted. This edit script always correctly transfers the original AST into the new AST, even if both ASTs are exactly the same. The actions of such an edit script would be correct but not accurate. For this paper we propose the following definition of an accurate edit action: An accurate action has to fulfill all of the following three criteria:

1) The action has to be comprehensible: why did the approach match the given nodes?
2) The action has to be helpful: did the action help to further the understanding of the changes occurring in the revision?
3) There can be no simpler solution: is there no comprehensible and helpful way to describe the change with fewer actions?

Applying GT and MTD to 307,081 Java source file revisions and manually analyzing a random sample of 2400 edit actions, we found that over 55% of GT's and over 81% of MTD's generated move and update actions are inaccurate according to our definition given above. We argue that such a high misclassification rate significantly impacts the understanding of large source code changes and in particular of moved and updated code.

With IJM we aim to reduce this misclassification rate and to generate edit scripts that are easier to understand and better reflect the original developer's intent. The improvements of IJM include partial matching, name-aware matching, and merging name nodes. Partial matching decreases the amount of nodes that are matched between different methods. Name-aware matching takes the names and values of nodes into account. Merging name nodes decreases the AST size by

merging some node types with their respective simple name nodes.

We evaluate IJM by comparing it to two state-of-the-art approaches, GT and MTD, answering the following four research questions.

RQ1: To which extent does IJM change the edit script size?
RQ2: What is the runtime of IJM to produce the edit scripts?
RQ3: To which extent does IJM reduce the misclassification rate of move and update actions?
RQ4: Can IJM be considered more helpful in terms of understanding the changes occurring in a revision?

The results show that IJM provides significantly better accuracy for move and update actions and is more beneficial to understanding the occurring changes. Also, IJM shows no increase of runtime or edit script size.

The remainder of the paper is organized as follows: Section II introduces AST differencing and points out the shortcomings of GT and MTD. Section III presents IJM and Section IV reports the results of its evaluation with 10 Java open-source projects. Section V discusses IJM's limitations and possible threats to validity of our results. Section VI gives an overview of the related work and, finally, Section VII draws the conclusions and discusses future work.

II. AST Differencing

ASTs are a tree-based representation of source code used by compilers, for example, in semantic analysis. ASTs consist of nodes with the following properties:

- A parent node: with the exception of the root node, every node has a parent.
- A label: the label represents the type of a node (*e.g.*, method declaration).
- A value: the value describes the content information of a node (*e.g.*, name of a class or method) and can be null.

ASTs can be of different granularity. For instance, a node could be as coarse as a whole statement or as fine grained as a single literal. For our purposes a finer granularity is preferable, since it allows the extraction of a detailed edit script. To derive an edit script between two different ASTs, most algorithms share the same two-steps approach. In the first step (matching) the algorithm matches similar nodes from both ASTs. A node can only belong to one match and only nodes with the same label can be matched. In the second step, differencing, an edit script is generated based on the matched and not matched nodes. There are optimal algorithms for the second step, such as presented by Chawathe *et al.* [12]. Therefore, we focus solely on the matching step in this paper.

The GT algorithm performs the matching in two phases: top-down and bottom-up. First, in the top-down phase, GT searches for isomorphic subtrees in both ASTs. The second phase, the bottom-up matching, takes the detected isomorphic subtrees as input. It matches parent nodes that share a significant number of matching descendants (controlled by three thresholds). GumTree then tries to match previously unmatched descendants of those nodes.

In contrast to GumTree, MTD is an AST differencer based on the approach of ChangeDistiller. It incorporates five optimizations as presented by Dotzler *et al.* in [5]. First, the identical subtree optimization reduces the matching problem by removing unchanged subtrees from the ASTs before applying a matching algorithm. This optimization is also used by GT. The other four optimizations all try to detect more moves to shorten the edit script. As shown by our manual analysis, this works for small changes, but leads to unnecessary move actions in the case of larger changes.

The following example, taken from the Apache Commons Lang project on GitHub, illustrates some of the problems of GT and MTD and shows some of the improvements provided by IJM. In Figures 1, 2, and 3, two consecutive versions of the `UnescapeUtils` class are shown. The changes introduced in the example (see Figure 1) are:

1) The static field `UNESCAPE_JAVA_CTRL_CHARS` is inserted (line 2).
2) `UNESCAPE_JAVA_CTRL_CHARS` is added as argument to the `AggregateTranslator` constructor call (line 15).
3) Five of the String arrays are moved from the `UNESCAPE_JAVA` field (lines 10–14) to the `UNESCAPE_JAVA_CTRL_CHARS` field (lines 5–9).

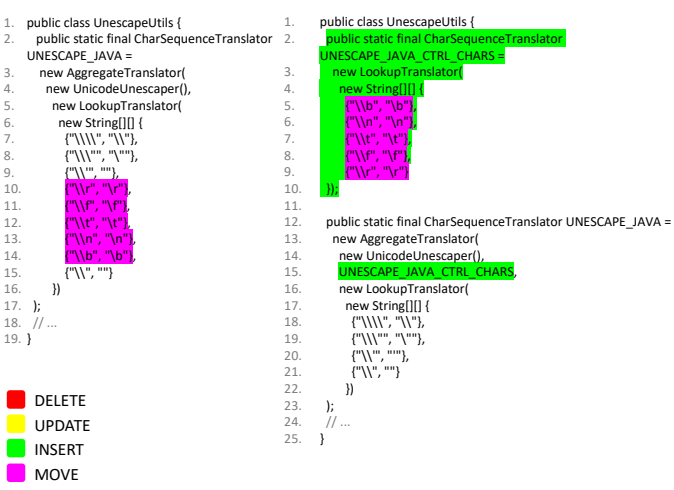

Fig. 1. Edit script generated with IJM. The highlighted parts show the differences between the two versions of the source file implementing the class `UnescapeUtils` of Apache's Commons Lang library.

The edit script for IJM, depicted in Figure 1, accurately represents all the changes listed before. In contrast, the edit script generated by GT is depicted in Figure 2, shows notable differences compared to Figure 1. GT matches the field `UNESCAPE_JAVA` from the original version with the field `UNESCAPE_JAVA_CTRL_CHARS` from the new version instead of `UNESCAPE_JAVA`. Through this inaccurate match, several additional edit actions are produced, such as the updates of the String literals in lines 7 to 14 of Figure 2. This is due to the lack of name awareness in GT's bottom-up phase.

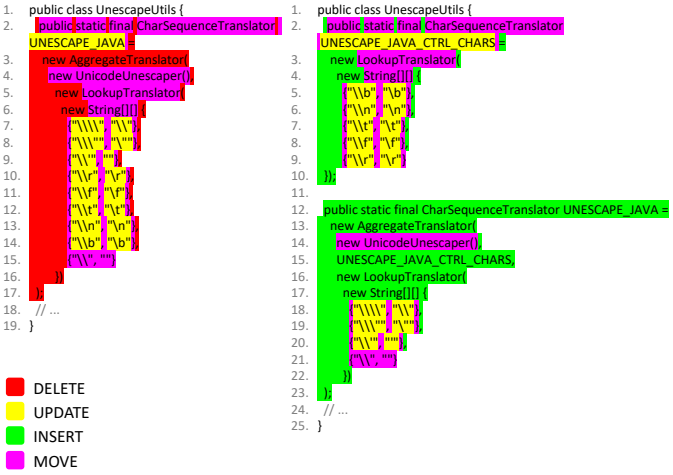

Fig. 2. Edit script generated with GT for the same revision as in Figure 1.

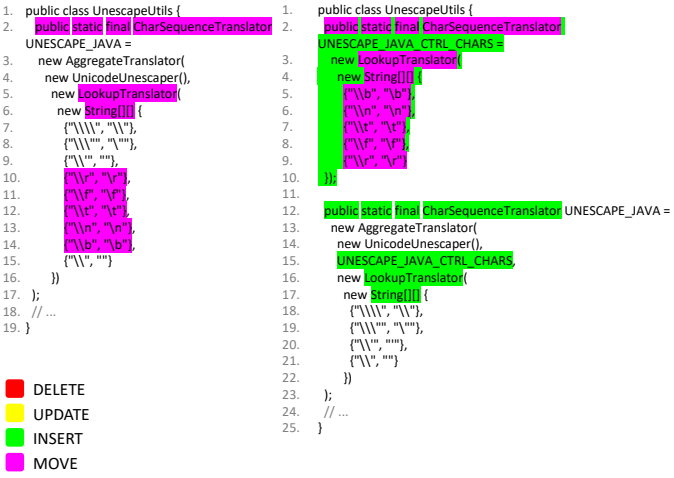

Fig. 3. Edit script generated with MTD for the same revision as in Figure 1.

The result generated by MTD, as depicted in Figure 3, is closer to the one by IJM but exemplifies a problem of MTD. `UNESCAPE_JAVA_CTRL_CHARS` is correctly detected as new field, but many nodes that are still present in the `UNESCAPE_JAVA` field declaration are moved to the new field declaration, introducing move and insertion edits. For instance, the modifier `public` is moved from `UNESCAPE_JAVA` to `UNESCAPE_JAVA_CTRL_CHARS` and the modifier `public` for the new `UNESCAPE_JAVA` is detected as insert.

The results of this small example show the main problem of GT and MTD: creating accurate move and update actions. Furthermore, MTD suffers from an additional problem – as stated by Dotzler *et al.* [5] it has problems processing large files whose ASTs contain more than 20,000 nodes.

III. IJM Approach

Our approach is based on the GT approach [4] and aims at improving the *matching* phase. Like GT and MTD, we use the existing implementation of the algorithm by Chawathe *et al.* [12] to create the final edit scripts. For improving the matching of AST nodes and subtrees, we add the following three matching strategies: 1) partial matching, 2) name-aware matching, and 3) merging of name nodes. In the following, we describe each matching strategy in detail.

A. Partial Matching

As shown in the previous section, both, GT and MTD, have difficulties accurately detecting moved and updated source code. Therefore, the main goal of IJM is to reduce the number of inaccurately classified moves and updates without increasing the misclassification rate for inserts and deletes. With partial matching we restrict the scope for the matching to selected parts of the source code. As we focus on Java source code, these parts are: import statements, type declarations, enumeration declarations, method declarations, and field declarations.

Each such part is represented by a subtree in the AST. Each of these subtrees has their own root node, referred to as subtree root node (SRN) to avoid confusion with the root node of the entire AST. We assume that most of the changes happen within such a subtree and only few changes happen between them. For instance, the source code of a method is more likely to be changed and moved within the same method. Furthermore, source code from a field declaration is not likely to be moved to a method declaration.

The partial matching approach is therefore to divide the AST into smaller parts that are individually matched. IJM is built of different matchers. A matcher takes two ASTs or subtrees as input and outputs a set of matched nodes. Composite matchers are collections of matchers that are applied subsequently. IJM itself is a composite matcher. General matchers can operate on any kind of AST while partial matchers address specific parts of the AST.

Figure 4 shows an overview of the matchers used by IJM and their configuration. Furthermore, it also shows the sequence (from left to right) in which the various matchers are applied. With partial matching we aim at identifying correct SRN node pairs to generate more accurate and compact edit scripts. In the following, partial matching is detailed, then each matcher is described separately.

Every partial matcher takes the following input: the source and destination ASTs, the mapping of all (previously) matched nodes, a *pruning condition*, and an *inner matcher*. It can additionally have a subtree matcher and a subtree *pruning condition*. As a result, each partial matcher adds the newly matched nodes to the list of previously matched nodes. The following three steps are performed by a partial matcher:

1) *Pruning:* This first step generates copies of the source and destination ASTs used by the partial matcher and removes all nodes from them that do not match the given *pruning condition*. Note, each partial matcher must specify such a condition. *Pruning conditions* can make use of all properties modeled in the AST such as node type, node position, children and parents. The *field declaration matcher* described below, for example, prunes all non-root nodes that are children of type declaration nodes and that are not field declaration nodes themselves. In

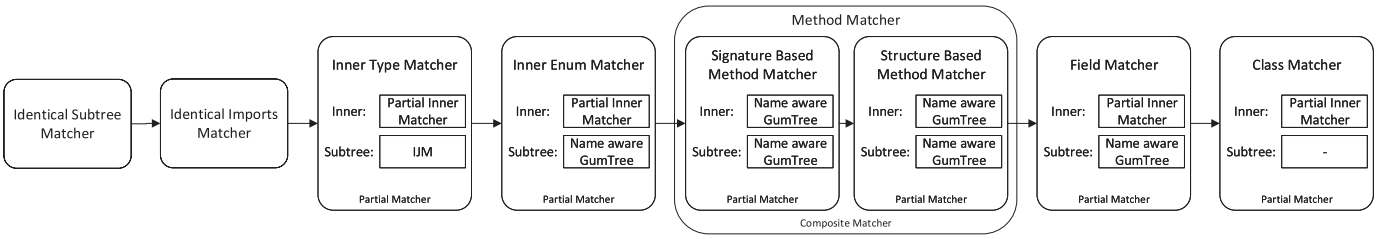

Fig. 4. Overview of the matchers used by IJM and the sequence (from left to right) of their application to the source and destination ASTs.

addition to the nodes that fit the condition, nodes that have already been matched by any previous matcher are also removed from the trees.

2) *Matching:* In the second step, the reduced ASTs are processed by an *inner matcher*. This can be any matcher and has to be specified in the matcher configuration. The matched nodes are then added to the mapping of all matched nodes of the original AST. If no further configuration is provided, this set represents the final result of the matcher. If subtree matching is enabled for a partial matcher, the algorithm continues with Step 3.
3) *Subtree Matching:* This phase matches the subtrees of the SRNs that have been matched in the previous step. Again, the specific matcher used has to be specified in the configuration. Most partial matchers use a variant of GumTree to match subtrees. All nodes matched in this process are added to the mapping of all matched nodes of the original AST.

IJM consists of a list of different matchers that are shown in Figure 4. In the following, we describe each matcher in the sequence of their application and its configuration.

- *Identical Subtree Matcher*: This general matcher is also used by GT and MTD and is described by Dotzler *et al.* [5]. It matches subtrees that exist in an identical form in both, the source and destination ASTs. It is not a partial matcher and therefore capable of and also used by IJM for detecting changes in the whole AST. This first matcher reduces the number of nodes that need to be processed by all matchers that are executed afterwards.
- *Identical Imports Matcher*: This matcher is a Java specific matcher that matches import statements. Any import statement that exists in the source as well as in the destination AST is matched. Similar to partial matchers, nodes that do not represent import statements are ignored. However, the identical imports matcher is not a partial matcher since no *inner matcher* can be specified.
- *Inner Type Matcher*: This is a partial matcher. The *pruning condition* of this partial matcher ensures that only subtrees of inner (non-anonymous) classes are processed. It uses the *Partial Inner Matcher*, a simplified version of GT detailed later, as inner matcher. If matching inner class declarations are found, the corresponding subtrees are processed using subtree matching as described in Step 3 before. This matcher uses IJM recursively as subtree matcher.
- *Inner Enum Matcher*: The inner enum matcher works similar to the inner type matcher. Instead of inner class nodes, enumeration classes declared within a Java class are processed by this matcher. The *Partial Inner Matcher* is used as inner matcher and the *Name-Aware GumTree* is used as subtree matcher.
- *Method Matcher*: This matcher processes all methods in the source and destination ASTs. As the majority of changes can be found within method bodies, it is crucial to correctly identify matching SRNs. The method matcher is a composite matcher consisting of two partial matchers applied subsequently: first, it attempts to match methods only by their signature, then it attempts to match the remaining methods by their signature and by their method body.
- *Signature-Based Method Matcher*: This partial matcher matches method declarations according to their signature. All nodes that are not part of the method signature are pruned. The *Name-Aware GumTree* (see Subsection III-C) is configured as inner and as Subtree Matcher.
- *Structure-Based Method Matcher*: Method declarations that have not been matched according to their signature are matched according to their structure. This matcher takes all nodes of the method declaration and body into account. The *Name-Aware GumTree* is configured as inner and as Subtree Matcher.
- *Field Declaration Matcher*: Any field that is declared within a given class is processed by this matcher. In contrast to the inner type and inner enum matcher, it does not process the subtrees of field declaration nodes. Instead, all field declaration subtrees are processed in a single iteration and the resulting matches are included in the final result.
- *Class Declaration Matcher*: This partial matcher is configured to match changes that occurred in class declarations. It uses the *Partial Inner Matcher* as inner matcher and does not use subtree matching.
- *Partial Inner Matcher*: This composite matcher is only used as inner matcher of partial matchers. It consists of two general matchers: The *Identical Subtree Matcher* described above and the *Name-Aware BottomUp Matcher* that is a modification of the same bottom-up matcher used by GT and is described below in Subsection III-C.
- *Name-Aware GumTree*: This composite matcher is a modified version of GT. Just as the *Partial Inner Matcher* it replaces the standard bottom-up matcher of GT with the modified *Name-Aware BottomUp Matcher* described

below in Subsection III-C.

With partial matching, nodes can only be matched if their SRNs have been matched. Moves between unmatched SRNs can therefore not be detected. For instance, a move from a method body to a different method body cannot be detected by partial matching. But note, moves within the same method body are detected. For the same reason, a partial matcher does not detect moves between nodes that have different types of SRNs, since those nodes cannot be matched. For example, partial matchers are unable to detect source code that has been moved from a method to a field declaration. However, when software developers refactor existing code, such changes are likely to happen. To address this specific problem and allow IJM to detect these changes, IJM uses the *Identical Subtree Matcher* before applying any partial matcher. This first matcher allows to identify and match moved code blocks in the source and destination ASTs.

B. Merged Name Nodes

Both, GT and MTD, run into problems with name nodes in the AST. Name nodes are children of various other nodes like method or type declarations containing the name of their parent node as value. The problem is that GT and MTD can match these name nodes with other name nodes having a different parent. Both matchers, as well as IJM, do not allow nodes of different types to be matched. However, since the name nodes share the same type (*simple name*), they can be matched even if their respective parents do not share the same type. The name of a method can therefore be matched to the name of a variable or class. Figure 5 provides an example of such a match: the name node of the method declaration `foo()`, that has been renamed to `bar()`, is (inaccurately) matched with the new variable declaration `foo`.

```
1. public class Test {
2.     public void foo() {
3.     }
4. }
```

```
1. public class Test {
2.     public void bar() {
3.         int foo = 1;
4.     }
5. }
```

Fig. 5. Example edit script generated using MTD where the name node `foo` has been moved from a method parent to a variable parent.

To prevent this, IJM modifies the AST by merging the value of name nodes with their respective parent nodes and deleting the name nodes with redundant information. The benefit of this modification is threefold: first, it prevents errors caused by moving the name node away from the parent node by forging an atomic node; second, it reduces the edit script size; and third, it decreases the length of the AST and therefore improves the runtime of our approach.

The tree on the left hand side of Figure 6 presents an excerpt of the AST generated by GT and MTD from the source code of the class `UnescapeUtils` shown in the Figures 1 to 3. The marked nodes are of the simple name type that are removed or merged with their respective parent nodes by IJM. The tree on the right hand side of Figure 6 shows the AST for the same source code created and used by IJM. The nodes that have been merged with their name nodes are marked. For instance, the simple name node `UnescapeUtils` is merged with its class declaration node. Another example is the simple name node of the static field `UNESCAPE_JAVA` that is merged with its variable declaration node. Furthermore, the simple name node `LookupTranslator` is deleted since it is already represented by its parent node. Regarding this example, IJM deleted 4 and merged 2 name nodes in the AST. This decreased the size of the AST in this excerpt from 20 to 14 nodes.

C. Name-Aware Matching

During our manual analysis of the edit scripts output by GT and MTD, we found several examples in which nodes with different names were inaccurately matched. Such an example is presented in Figure 7 in which the nodes representing `fields` and `iterator`, and the nodes representing `isEmpty` and `hasNext` are matched. While the two method invocations are structurally equivalent, they clearly differ semantically as indicated by their different names. The resulting edit script contains unnecessary move and update actions (instead of only representing the change with one delete and one insert action). The reasons for these matches are, that GT, in the bottom-up phase, only uses the node type to determine whether or not two nodes can be matched. Consequently, because the structure of the subtrees of the `if`-condition and the node types are equal, the nodes mentioned above are matched.

IJM addresses this problem by adding name-awareness to the bottom-up phase of GT. This is realized by considering the similarity of the names of the nodes in addition to their node types. The similarity is computed using the weighted Levenshtein distance [13]. We experimented with different values and decided that two names are considered similar if they have a distance of < 0.3. This similarity threshold is introduced for nodes that represent: method declarations, method invocations, enum declarations, enum constant declarations, import declarations, and name nodes. Referring to the previous example, IJM does not match the nodes representing `fields` and `iterator`, and the nodes representing `isEmpty` and `hasNext`. The result is an edit script where the condition of the `if` statement is completely removed and replaced by a new condition statement.

IV. Evaluation

For the empirical evaluation of IJM we compared it to GT and MTD. As described in Section III, IJM is based on GT but adds partial matching, name-awareness, and works on modified ASTs. For this evaluation we propose evaluating four different criteria: Edit script size, Runtime, Edit action accuracy, and Edit script helpfulness.

Dotzler and Philippsen [5] show that shorter edit scripts are more helpful to understanding changes between revisions. We therefore use edit script size as a metric. For the approaches to be of practical use, they have to run in a reasonable amount of time, thus we compare the runtime for matching and creating the edit scripts for a revision. Edit action accuracy has been

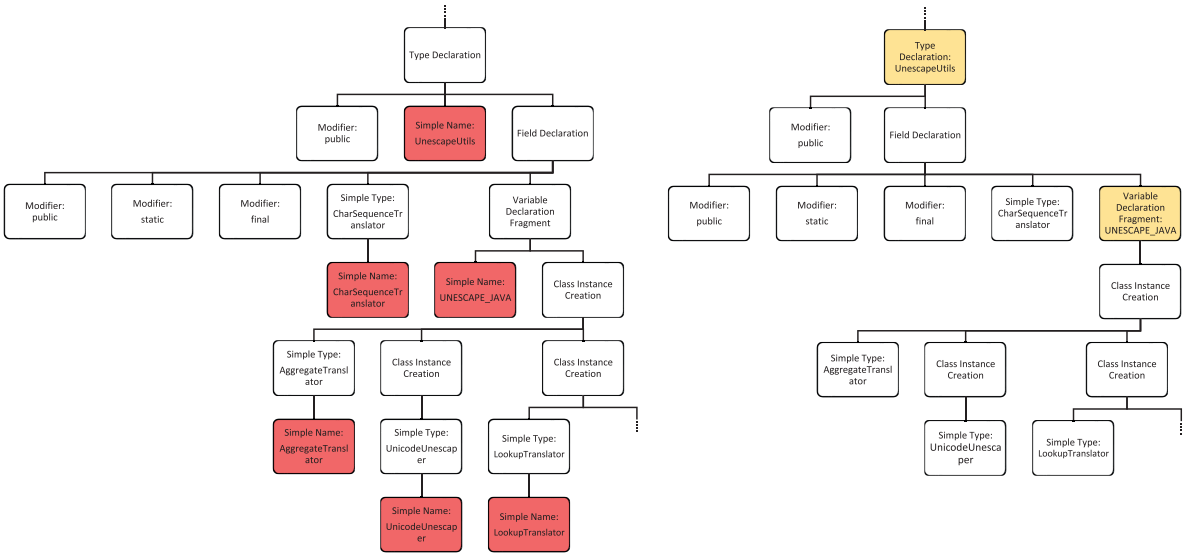

Fig. 6. Tree on the left shows the excerpt of the AST generated by GT and MTD for the source code of the class `UnescapeUtils` (see Figure 1). Tree on the right shows the same excerpt of the AST generated by IJM. Marked nodes represent simple name nodes that are merged or removed by IJM.

```
1. public class Test {
2.   public void foo() {
3.     if (fields.isEmpty()) {
4.       //impl
5.     }
6.   }
7. }
```

```
1. public class Test {
2.   public void foo() {
3.     if (iterator.hasNext()) {
4.       //impl
5.     }
6.   }
7. }
```

Fig. 7. Example edit script generated using GT where the nodes representing `fields` and `hasNext`, and `isEmpty` and `hasNext` are inaccurately matched.

added to measure whether or not the single actions of an edit script accurately depict the changes in a given revision. As fourth criterium, we evaluate the helpfulness of complete edit scripts to see if they further the understanding of the occurring changes in a revision.

We use 10 well-known open source Java projects as data set for the evaluation as can be seen in Table I. We chose these projects to cover a broad bandwidth of projects of different sizes, ranging from 269 to over 10,000 classes and from 1,327 to 17,948 commits. All projects are open-source and publicly available from GitHub to ensure reproducibility. We ran the three approaches IJM, MTD, and GT on all file revisions from all non-merge commits of those 10 projects to generate the edit scripts. 11,353 out of 392,492 (2.89%) revisions could not be handled by MTD with our setup, since the process ran out of memory (we ran MTD with dedicated 40 GB of RAM). Neither GT nor IJM ran into this problem. To allow for a fair comparison of the approaches, we excluded these revisions from the data set. We also excluded any revisions that have changes in JavaDoc (76,785), since IJM focuses on source code changes and is not able to detect JavaDoc changes. Using this process, we generated a total of 307,081 edit scripts per approach. Note that IJM ran on the reduced AST whilst GT and MT ran on the larger unmodified AST.

TABLE I
DESCRIPTIVE STATISTICS OF THE 10 JAVA OPEN SOURCE PROJECTS USED IN THE EVALUATION

Poject	Commits	Revisions	LOC	Methods	Classes
ActiveMQ	7,413	44,829	405,747	41,730	4,940
Commons IO	1,327	4,443	29,267	44,448	269
Commons Lang	3,742	11,034	74,477	51,062	539
Commons Math	5,010	32,132	186,566	65,695	1,646
JDT Core	3,658	26,884	1,400,678	137,155	7,842
HBase	17,948	89,119	1,116,946	254,346	8,491
Hibernate ORM	10,097	63,393	643,299	321,547	10,758
Hibernate Search	6,002	87,465	137,468	335,372	2,576
JUnit 4	1,376	6,276	28,749	339,376	1,145
Spring Roo	4,467	26,917	106,454	347,608	998
All	61,040	392,492	4,129,651	1,938,339	39,204

TABLE II
MEDIAN AND MAXIMUM EDIT SCRIPT SIZE PER PROJECT AND APPROACH

Project	Median			Maximum		
	GT	MTD	IJM	GT	MTD	IJM
ActiveMQ	13	13	10	20,171	20,173	15,313
Commons IO	10	10	8	2,273	3,276	1,847
Commons Lang	13	13	10	4,494	4,494	3,912
Commons Math	7	8	6	8,332	8,334	6,442
JDT Core	17	18	14	30,532	30,552	22,344
HBase	11	11	8	76,057	76,059	59,932
Hibernate ORM	9	11	6	27,605	27,607	20,250
Hibernate Search	8	8	6	2,543	2,539	1,947
JUnit 4	13	14	10	2,000	2,002	1,530
Spring Roo	13	13	10	3,830	4,160	3,093
All	12	12	9	76,057	59,932	76,059

A. Edit Script Size

Existing work on tree differencing, such as [4], [5], and [14], argue that smaller edit scripts are better in terms of understandability. Table II presents the results in terms of median and maximum edit script size per project and approach. The minimum edit script size for each approach and project is 1.

When used on the 307,081 revisions from the data set, IJM produces edit scripts, whose median size is 9. The median

edit scripts size of GT and MTD is 12. Looking at the values per project, we observe that IJM produces shorter median edit scripts for all of the 10 Java projects. In 95.22% of all evaluated revisions, IJM produced the smallest edit script either alone or shared first place with one of the other two approaches. This holds true for 53.08% of all revisions for GT and for 54.53% respectively for MTD.

Based on these numbers, we checked whether the differences in the median values are statistically significant using the Wilcoxon signed rank test ($\alpha < 0.01$) and Cliff's Delta d [15]. We used the Wilcoxon singed rank test and Cliff's Delta since the sizes of the edit scripts are non-normal distributed. Concerning Cliff's Delta, the effect size is considered negligible for $d < 0.147$, small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.47$, and large for $d \geq 0.47$ [16]. The p-values of the Wilcoxon tests for comparing the size of all edit scripts of GT and IJM and for MTD and IJM are $< 2.2e^{-16}$. The effect size d over all edit scripts between GT and IJM is 0.084 and between MTD and IJM is 0.102. Both are therefore considered negligible. Performing the comparison of the edit script sizes per project, we obtained p-values $< 6.01e^{-169}$ for the Wilcoxon tests. The corresponding values for the Cliff's delta except one are all < 0.147. Only in the case of comparing MTD and IJM on Hibernate ORM, Cliff's delta is 0.172 thus considered small. This shows that while the median sizes of the edit scripts output by the three approaches differ significantly, the differences can be considered negligible and at most small.

Based on these results, the research question RQ1 "To which extent does IJM change the edit script size?" can be answered as follows: IJM, run on the reduced AST, reduces the median edit script size of GT and MTD by 3 edits from 12 to 9.

B. Runtime

Approaches that produce small and accurate edit scripts might still not be the best option to use if they do not complete in a practicable amount of time. We therefore evaluate and compare the runtime of IJM, GT, and MTD. For all three approaches we measure and compare the following data: Matching time (MT), Action generation time (AGT), Total time for revision (TTR).

MT describes the time used to solely match all nodes of a given revision, not including parsing the two ASTs. For IJM this includes the time used by the identical subtree matcher as well as all partial matchers. AGT describes the time used to solely generate the set of edit actions after matching the nodes of a given revision. TTR describes the total time being used to process a revision. This includes the time used to generate the AST, the MT, the AGT, and the possible overhead such as the AST modification of IJM. All values are given in milliseconds (ms).

To generate comparable data, we ran 3 iterations of every approach, each on a separate PC, on every revision from the data set. We enforced an artificial time limit of 3 minutes per revision. If any approach ran out of memory or hit the time limit the revision was excluded from this part of the evaluation for all approaches. This only happened in 47 cases, all by MTD. This process generated a total of 921,102 data points for each approach. All three PCs used for the evaluation have the exact same specification.[1] The results of the runtime evaluation are shown in Tables III–V. Table III depicts the median and maximum runtime of the matching part (MT) of the three approaches. Table IV depicts the median and maximum runtime of the action generation part (AGT) and Table V depicts the total runtime (TTR) of the three approaches. The median values for MT show that IJM outperforms GT in 6 and ties with it in 4 out of 10 projects. In comparison to MTD, IJM outperforms it in 8 out of 10 projects and ties in 2. The median

[1]Dell OptiPlex 9020 Ultra Small Form Factor, Intel Core i5-4590S (Quad Core, 6MB, 3.00GHz w/HD4600 Graphics), 16GB RAM

TABLE III
MEDIAN AND MAXIMUM RUNTIME FOR THE MATCHING (MT) IN MS PER REVISION AND APPROACH

Project	Median			Maximum		
	GT	MTD	IJM	GT	MTD	IJM
ActiveMQ	2.67	3.00	2.00	3,960.00	11,063.00	1,350.33
Commons IO	2.33	3.67	2.33	361.00	6,106.33	629.67
Commons Lang	4.67	10.00	4.67	930.33	17,883.00	240.67
Commons Math	2.67	3.67	2.67	9,881.00	19,633.00	705.67
JDT Core	6.33	16.67	6.00	42,208.33	144,270.67	8,165.67
HBase	14.00	26.33	7.00	3,177.00	133,944.33	9,941.67
Hibernate ORM	1.33	1.33	1.33	1,121.00	27,131.67	4,164.33
Hibernate Search	2.33	2.00	2.00	2,347.67	48,007.67	3,194.67
JUnit 4	2.00	2.00	1.67	466.00	294.67	1,819.00
Spring Roo	6.00	5.00	3.33	1,995.33	8,900.33	2,714.00
All	4.00	4.33	3.00	42,208.33	144,270.67	9,941.67

TABLE IV
MEDIAN AND MAXIMUM RUNTIME FOR THE ACTION GENERATION (AGT) IN MS PER REVISION AND APPROACH

Project	Median			Maximum		
	GT	MTD	IJM	GT	MTD	IJM
ActiveMQ	0.67	0.67	0.33	62.33	123.67	45.67
Commons IO	0.67	0.67	0.67	52.00	47.67	34.33
Commons Lang	1.67	1.67	1.00	157.67	100.00	108.00
Commons Math	0.67	0.67	0.67	385.33	405.67	370.00
JDT Core	1.67	1.67	1.33	6,277.00	5,690.00	6,160.33
HBase	1.33	1.67	1.00	1,728.67	3,403.67	1,127.33
Hibernate ORM	0.33	0.33	0.33	117.33	245.00	68.67
Hibernate Search	0.33	0.33	0.33	29.00	36.33	17.00
JUnit 4	0.33	0.33	0.33	14.33	16.33	10.67
Spring Roo	0.67	0.67	0.67	118.00	119.67	88.33
All	0.67	0.67	0.67	6,277.00	5,690.00	6,160.33

TABLE V
MEDIAN AND MAXIMUM TOTAL RUNTIME (TTR) IN MS PER REVISION AND APPROACH

Project	Median			Maximum		
	GT	MTD	IJM	GT	MTD	IJM
ActiveMQ	6.00	6.00	5.00	4,007.00	11,230.66	1,387.33
Commons IO	6.67	8.00	6.33	458.00	6,233.00	708.66
Commons Lang	13.33	20.00	12.00	957.67	18,104.33	809.00
Commons Math	7.00	8.00	6.33	9,891.33	20,115.67	1,196.33
JDT Core	15.00	26.00	13.00	42,329.00	145,095.67	8,448.66
HBase	22.66	36.33	14.33	4,171.33	134,404.33	10,007.00
Hibernate ORM	3.67	3.33	3.33	1,158.33	27,365.67	4,241.33
Hibernate Search	5.33	4.67	4.00	2,364.67	48,025.00	3,208.00
JUnit 4	4.67	4.33	4.00	564.33	567.33	1,823.33
Spring Roo	11.00	9.00	7.00	2,032.00	9,029.00	2,784.00
All	8.33	9.00	6.67	42,329.00	145,095.67	10,007.00

values for AGT show that the three matchers behave similarly. In 4 out of 10 projects IJM outperforms GT and MTD and ties in 6. The median values for TTR show that IJM outperforms GT and MTD in 9 out of the 10 projects. IJM reduces the median runtime from 8.33 ms and respectively 9 ms to 6.67 ms. Only in the Hibernate ORM project IJM and MTD tie with a median TTR of 3.33 ms.

We checked whether the differences in the median values are statistically significant using the Wilcoxon signed rank test ($\alpha < 0.01$) and Cliff's Delta d [15]. The p-values of the Wilcoxon tests for comparing the TTR over all revisions of all projects between GT and IJM and between MTD and IJM are both $< 2.2e^{-16}$. The effect size between GT and IJM is 0.06 and is considered negligible. The effect size between MTD and IJM is 0.116 and also considered negligible. We obtained similar results for comparing the results for MT and AGT over all revisions and all projects. All the p-values show a significant difference in the median values while the values of Cliff's delta show that these differences are negligible. Computing the statistics for each project, we found small effect sizes in MT and TTR for Commons Lang, JDT Core, and HBase between IJM and MTD. Furthermore, for HBase we found also a small effect size for AGT between IJM and MTD, as well as a small effect size for MT between IJM and GT. The values of Cliff's delta for all other projects showed a negligible effect. All computed statistics are significant with p-values $< 5.2e^{-13}$.

Hence, the research question RQ2 "What is the runtime of IJM to produce the edit scripts?" can be answered as follows: IJM, run on the reduced AST, reduces the median runtime to process a single revision compared with GT and MTD from 8.33ms and respectively 9ms to 6.67ms.

C. Edit Action Accuracy

Previous research, such as [5], has mostly been focused on edit script size and runtime to evaluate matching approaches. In some studies, the quality of the generated edit scripts has been evaluated but the data set was highly constrained (*e.g.*, only one change per script) [4] or the data set was small [3]. While an evaluation of edit script size and runtime is absolutely necessary, we add the level of edit action accuracy to the evaluation to get a better understanding of the generated edit scripts' quality.

Evaluating the quality of an edit script is no trivial task – some actions of an edit script can be beneficial for the understanding of the underlying change in the file while other actions in the same revision might be misleading. Therefore, we evaluated both, whole edit scripts (Section IV-D) as well as the accuracy of single edit actions. For this evaluation we randomly selected 2400 single edit actions from our data source (200 for each of the 4 action types and for each approach).

All actions have been independently classified as either accurate or inaccurate according to the definition given in Section I by two evaluators that are also co-authors of this paper and know the details of the three matching approaches.

TABLE VI
MISCLASSIFICATION RATE PER MATCHER. MR DENOTES THE MISCLASSIFICATION RATE, NOA DENOTES THE NUMBER OF ACTIONS

Action	GT		MTD		IJM	
	MR	NoA	MR	NoA	MR	NoA
MOVE	58.2%	720,303	81.5%	3,121,607	43.5%	510,250
UPDATE	40%	938,288	37%	759,177	17%	503,423
INSERT	5.5%	12,225,111	6%	9,642,897	5.5%	10,236,135
DELETE	12%	5,478,973	11%	4,038,471	11.5%	5,021,193
Relative MR:	10.98%		21.91%		8.9%	

If both evaluators classified a given action as either accurate or inaccurate the action was classified as such. All cases where the two evaluators differed in their classification were discussed in a second iteration. This second iteration resulted in no disagreements.

The results of the classification are depicted in Table VI. They show that especially move and update actions pose a problem to all three approaches. As shown by the values for MOVE, more than half of GT's and more than 80% of MTD's move actions have been classified as inaccurate, while IJM manages to reduce this number to 43.5%. The IJM approach also improves the accuracy of update classifications without increasing the misclassification rate of insert or delete actions compared to both GT and MTD. However, there is a vast imbalance concerning the distribution of the action types. For example, only 3.7% (GT) or 3.1% (IJM) of all actions are moves. MTD on the other hand generates 17.8% moves and shows the highest misclassification rate for them. Thus, it is important to put the misclassification rate in relation with the distribution of edit action types. The resulting relative misclassification rate of IJM is reduced to 8.9% compared to GT's 10.98% and MTD's 21.91%.

Based on these results we can answer RQ3 "To which extent does IJM reduce the misclassification rate of move and update actions?" with: Compared to GT and MTD, IJM reduces the misclassification rate of move actions by 14.7% and respectively 38.0%. The misclassification rate of update actions is reduced by 23% compared to GT and by 20% compared to MTD.

D. Edit Script Helpfulness

Since the previous evaluation has been performed by two co-authors of this paper, it could involuntarily be biased. Therefore, we conducted a second experiment with 11 independent external experts evaluating the helpfulness of the edit scripts in terms of understanding the changes present in the revision. All participants of the study are professional software developers or researchers holding at least a masters degree in computer science. All of them work with versioning systems and know the textual diff representations used by such systems.

The experiment was set up as follows: we randomly selected 3 revisions from every project in the data source. Each revision had to consist of between 20 and 100 edit actions and include at least one move or update action. We also excluded all revisions with no difference in the edit scripts between the approaches. Each participant was presented with 10 revisions (one per project) and the three corresponding edit scripts

generated by GT, MTD, and IJM. We used an online evaluation tool, that visualizes the four types of edit actions (similar to Figures 1–3). In the case of move and update actions, this tool allows the user to view the corresponding nodes in the original and modified version of the source code to see which nodes have been matched. We anonymized the approaches and randomized the order in which the edit scripts were shown. All participants were given a short introduction into how to use the evaluation tool. Then, for each revision, we asked the participants to rank the three edit scripts according to their helpfulness in terms of understanding the changes present in the revision. Each revision has been ranked by at least 3 and maximum 5 experts resulting in 110 rankings.

Table VII shows how often each matcher has been ranked first place per revision. It also shows how often all matchers have been ranked first, second, and third place in total. IJM ranks first in 49 out of 110 cases (44.5%) and for 18 out of 30 revisions. In 13 out of these 18 revisions, IJM got ranked sole first. For 5 revisions, GT was ranked sole first, and MTD was ranked sole first for 7 out of the 30 revisions. For 3 revisions, IJM was ranked first together with GT, for 1 revision IJM was ranked first with MTD, and finally, for 1 revision all three matchers were ranked first by one of the participants. Comparing the counts for first, second, and third place obtained for the three matchers with the Pearson's Chi^2 test results in a Chi^2 of 9.55 with a p-value of 0.049. This shows a dependency between rankings and the matcher. Since the p-value is below 0.05 it is unlikely that the results occurred by chance.

Based on these results, we answer research question RQ4 "Can IJM be considered more helpful in terms of understanding the changes occurring in a revision?" with: our expert based evaluation shows that they found IJM more helpful than GT or MTD.

V. Discussion

In the following, we discuss the implications of our findings on research, the limitations of our approach, and threats to validity.

A. Implications

As discussed in Section I, a wide variety of approaches uses edit scripts as their basis. The results of our evaluation presented in Section IV show that such existing approaches and future applications should consider using IJM, because it creates more accurate (Section IV-C) and more helpful (Section IV-D) edit scripts without increasing the runtime or edit script size compared to GT and MTD. We assume that the differences in runtime and edit script size are due to merged name nodes and reduced AST size and that GT and MTD would therefore also profit from this approach. However, we did not evaluate their accuracy when run on a reduced AST, as we did not find any indications for a beneficial effect during a manual review. It should also be noted that whether or not an edit script can be considered accurate or helpful is depending on the context. For this paper we focused on the developers understanding of a change, other applications, such as clone detection, might not consider the edit scripts generated by IJM accurate. Additionally, future change extraction algorithms that intend to create not only correct but better understandable edit scripts should not set their sole focus on creating the shortest edit script possible but should take the developer's intent into account. The evaluation proposed in Section IV-D is not limited on being applied to IJM and can easily be adopted to evaluate future approaches.

B. Limitations and Threats to Validity

One threat to validity for this research is that the two evaluators of the edit action accuracy are also co-authors of this paper. To mitigate this bias, we conducted an additional experiment using 11 external experts, that did not have any knowledge about the approaches compared. In addition, we make the source code[2] and dataset that we used for the evaluation publicly available.[3]

Another threat concerns the generalizability of the results. In it's current state, IJM is limited to work only with Java source code, even though its approach can be easily adopted to fit other object-oriented programming languages. However, no conclusions about it's performance on other languages can be drawn from this evaluation.

It is possible that the implementations of IJM, GT, or MTD suffer from bugs. This is, however, not very likely, since none of the manually evaluated edit scripts showed any indications of bugs except the following shortcoming. During our analysis we found a shortcoming in the generation of the ASTs that affects the matching and generation of the edit scripts in all three approaches. For instance, if `y = a.bar(x);` would be changed to `y = bar(a,x);`, the AST created with Eclipse JDT and used by all three approaches does not differentiate the node types of `"a"`. Both are considered *simple name* nodes. Therefore the approaches produce wrong edit scripts in this case. GT and MTD match both `"a"`s despite their semantic differences, IJM produces no change in this specific case. Investigating all revisions, we found that this shortcoming affects at most 4.7% of the revisions and at most 0.3% of all changes in our data set, therefore, we view our results as sufficiently valid.

It is also possible that the performance problems of MTD are due to a suboptimal implementation and not due to the algorithm itself. The similarity thresholds for names in this study have been set to 0.3 for IJM and to 0.3 for GT. GT also implements a tree size threshold of 1,000 that has not been modified. MTD's leaf threshold has been set to 0.88, the weight similarity to 0.37, and the weight position to less than 0.0024. Neither the thresholds for GT nor for MTD have been modified from their implementation found on GitHub (https://github.com/GumTreeDiff/gumtree, https://github.com/FAU-Inf2/treedifferencing).

[2]https://github.com/VeitFrick/IJM

[3]https://github.com/IterativeJavaMatcher/IJM_Reproductionset

TABLE VII
RESULTS FROM THE 11 EXPERTS RANKING THE EDIT SCRIPTS OF 30 RANDOMLY SELECTED REVISIONS (R1–R30)

Approach	Sum of first place rankings per revision																														Rank sum		
	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	1st	2nd	3rd
GT	2	1	0	3	3	2	1	1	0	1	0	0	0	1	1	0	0	3	1	0	0	1	2	2	1	1	2	0	1	0	30	39	41
MTD	0	1	3	1	0	0	3	0	2	1	2	1	2	1	0	0	1	0	2	1	1	0	0	0	3	2	1	0	2	1	31	39	40
IJM	2	3	0	1	0	2	0	2	2	2	3	2	1	1	2	4	3	1	0	3	4	2	1	2	0	1	0	3	0	2	49	32	29

VI. RELATED WORK

Extracting changes between two versions of a source file is a problem that has been investigated by several researchers. We divide this previous research into three categories: textual or line-based differencing, tree-based differencing, and other approaches.

A. Textual Differencing

Textual differencing algorithms, such as the famous GNU diff, are based on the algorithms presented in [17] and [2], and compare source code on a purely textual level. These algorithms are capable of detecting inserted, deleted, and updated lines of text. Some newer algorithms, such as presented by Reiss in [18] and Canfora *et al.* in [19] are able to detect moved lines as well. All these approaches have in common that they do not use the syntax information contained in source code. This hinders the exact classification of detailed changes in source code, but in turn enables their application to any kind of text and therefore these approaches are mostly language independent.

B. Tree-Based Differencing

Unlike textual differencing, tree differencing does not compare the text of the two versions of source code but uses the underlying tree structure of source code to generate edit scripts that transform the source AST into the destination AST. There are many tree-based approaches generating edit scripts comprised of insert, delete, and update actions, but without move actions. Bille published a survey that provides an overview on such tree edit distance algorithms [20].

The algorithm introduced by Chawathe *et al.* [12] is capable of computing edit scripts containing update and move actions based on trees generated from LaTeX source files. Optimal algorithms, not considering moved subtrees, like RTED [21], exist and run in $O(n^3)$. ChangeDistiller [3], presented by Fluri *et al.*, is a tool that uses a reduced AST to extract and classify changes between two versions of a file using tree-differencing.

IJM itself builds upon GT [4] which uses AST differencing and is inspired by the algorithm of Cobena *et al.* [22]. MTD, short for Move Optimized Tree Differ, is another recent approach presented by Dotzler *et al.* [5] to optimize the GT and other matching algorithms. It tries to decrease the size of its generated edit scripts by increasing the number of moves but runs into problems concerning runtime and misclassification of move actions as we demonstrated in Section II.

Diff/TS [23] is another approach to generate edit scripts from ASTs. It supports multiple languages and is able to detect move actions. However, it has no evaluation in terms of quality and its source code has not been made publicly available, therefore it was not possible to include it in our evaluation. Recently, Higo *et al.* proposed an algorithm extending GT that also considers copy-and-paste actions in addition to insert, delete, update, and move actions [14]. While this is a valuable extension, it suffers from the same shortcomings as GT.

There are additional approaches, designed to work with a specific programming language or family of languages. For instance, VDiff [24] is designed to work with the hardware definition language Verilog. It uses an approach, similar to that of ChangeDistiller.

C. Other Approaches

Multiple other approaches use graphs to uncover differences between source code files. For instance, JDiff [25] is a Java specific tool that is based on enhanced control-flow graphs. UMLDiff [26] works on UML models of software represented in a graph structure. srcDiff [27] is a differencing approach using an XML format that embeds syntactic information in source code files [28]. Dex [29] uses abstract semantic graphs to create edit scripts. FMDiff [30] is an approach implemented with the EMF Compare framework to extract feature model changes from the Linux kernel files. CSeR [31], [32] is an AST based tool for clone differencing, it uses the same edit actions as IJM but computes them by incrementally updating changes between the clones.

VII. CONCLUSIONS

In this paper we presented IJM, an improved approach to generate edit scripts from different file versions using tree differencing. The improvements mainly focus on the matching phase and include partial matching, name aware matching, and merging name nodes. We evaluated and compared the accuracy and helpfulness of IJM to the state of the art approaches GT and MTD. A study with 2400 randomly selected edits shows a higher accuracy for move and update actions without increasing the misclassification rate for insert and delete actions (Section IV-C). A study with 11 independent external experts showed that they found IJM more helpful for understanding the changes in a revision (Section IV-D). Furthermore, an evaluation on 10 Java open source projects shows no increase in edit script size (Section IV-A) and runtime (Section IV-B).

In future work, we plan to further improve the matching by considering the type information of all name nodes. Furthermore, we plan to extend the taxonomy of ChangeDistiller to consider detailed changes within statements.

ACKNOWLEDGEMENT

This work has been funded by the Austrian Science Fund (FWF) under project number 2753-N33.

REFERENCES

[1] L. Torvalds and J. Hamano, "Git: Fast version control system," *URL http://git-scm.com*, 2010.

[2] E. W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, Nov 1986.

[3] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[4] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 313–324.

[5] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16, Sept 2016, pp. 660–671.

[6] 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 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '16. New York, NY, USA: ACM, 2016, pp. 511–522.

[7] C. Macho, S. Mcintosh, and M. Pinzger, "Extracting build changes with builddiff," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 368–378.

[8] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '17. New York, NY, USA: ACM, 2017, pp. 740–751.

[9] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, "Discovering bug patterns in javascript," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '16. New York, NY, USA: ACM, 2016, pp. 144–156.

[10] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 351–360.

[11] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 502–511.

[12] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '96. New York, NY, USA: ACM, 1996, pp. 493–504.

[13] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics-Doklady*, vol. 10, pp. 707–710, 1966.

[14] Y. Higo, A. Ohtani, and S. Kusumoto, "Generating simpler ast edit scripts by considering copy-and-paste," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 532–542.

[15] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, p. 494, 1993.

[16] R. J. Grissom and J. J. Kim, *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Erlbaum Associates, 2005.

[17] W. Miller and E. W. Myers, "A file comparison program," *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

[18] S. P. Reiss, "Tracking source locations," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 11–20.

[19] G. Canfora, L. Cerulo, and M. Di Penta, "Tracking your changes: A language-independent approach," *IEEE Software*, vol. 26, no. 1, pp. 50–57, 2009.

[20] P. Bille, "A survey on tree edit distance and related problems," *Theoretical Computer Science*, vol. 337, no. 1-3, pp. 217–239, 2005.

[21] M. Pawlik and N. Augsten, "Rted: A robust algorithm for the tree edit distance," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 334–345, Dec 2011.

[22] A. Marian, "Detecting changes in xml documents," in *Proceedings of the 18th International Conference on Data Engineering*, ser. ICDE '02. Washington, DC, USA: IEEE Computer Society, 2002, p. 41.

[23] M. Hashimoto and A. Mori, "Diff/ts: A tool for fine-grained structural change analysis," in *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, ser. WCRE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 279–288.

[24] A. Duley, C. Spandikow, and M. Kim, "A program differencing algorithm for verilog hdl," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 477–486.

[25] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ser. ASE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–13.

[26] Z. Xing and E. Stroulia, "Umldiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 54–65.

[27] M. J. Decker, "srcdiff: Syntactic differencing to support software maintenance and evolution," Ph.D. dissertation, Kent State University, 2017.

[28] J. I. Maletic and M. L. Collard, "Supporting source code difference analysis," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, Sept 2004, pp. 210–219.

[29] 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 *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ser. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 188–197.

[30] N. Dintzner, A. Van Deursen, and M. Pinzger, "Extracting feature model changes from the linux kernel using fmdiff," in *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS'14. New York, NY, USA: ACM, 2013, pp. 22:1–22:8.

[31] F. Jacob, D. Hou, and P. Jablonski, "Actively comparing clones inside the code editor," in *Proceedings of the 4th International Workshop on Software Clones*, ser. IWSC '10. New York, NY, USA: ACM, 2010, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/1808901.1808903

[32] D. Hou, F. Jacob, and P. Jablonski, "Exploring the design space of proactive tool support for copy-and-paste programming," in *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '09. Riverton, NJ, USA: IBM Corp., 2009, pp. 188–202. [Online]. Available: https://doi.org/10.1145/1723028.1723051