Generating Simpler AST Edit Scripts by Considering Copy-and-Paste

Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan

***Abstract*—In software development, there are many situations in which developers need to understand given source code changes in detail. Until now, a variety of techniques have been proposed to support understanding source code changes. Tree-based differencing techniques are expected to have better understandability than text-based ones, which are widely used nowadays (e.g., diff in Unix). In this paper, we propose to consider copy-and-paste as a kind of editing action forming tree-based edit script, which is an editing sequence that transforms a tree to another one. Software developers often perform copy-and-paste when they are writing source code. Introducing copy-and-paste action into edit script contributes to not only making simpler (more easily understandable) edit scripts but also making edit scripts closer to developers' actual editing sequences. We conducted experiments on an open dataset. As a result, we confirmed that our technique made edit scripts shorter for 18% of the code changes with a little more computational time. For the other 82% code changes, our technique generated the same edit scripts as an existing technique. We also confirmed that our technique provided more helpful visualizations.**

I. Introduction

In various situations in software development such as source code review and merge conflict resolution, developers have to understand given source code changes in detail. To support understanding source code changes, previous work has proposed various techniques that visualize given changes.

The most widely-used differencing techniques are text-based ones [1], [2], [3]. For example, in Unix diff, added lines and deleted lines are visualized with special prefixes. An issue of the text-based differencing techniques is that the structure of source code is not considered. Because of this, visualized changes are not necessarily easy to understand in detail.

Techniques overcoming the issue of the text-based differencing techniques are AST[1]-based ones [4], [5], [6], [7]. In AST-based differencing techniques, given changes are visualized according to the structure of source code. In the techniques, an **edit script** is generated from two ASTs which have been built from pre-change and post-change source code. An edit script is a sequence of editing actions to convert the pre-change AST to the post-change AST [4]. Longer edit scripts are generated from bigger changes. Falleri et al. showed that length of edit scripts could be an indicator of the need to understand given changes [8]. Changes with shorter edit scripts are easier to understand.

Generating an edit script takes a long time if target source code is large or code move is considered. Falleri et al. succeeded to shorten generation time by adopting some heuristics [8]. They also showed that their edit scripts were more helpful to understand the changes.

Currently, we are trying to generate more easily-understandable edit scripts. In this paper, we propose a new technique for edit script generation. Our technique is based on Falleri's one, which visualizes code changes with four kinds of actions: inserting, deleting, updating, and moving. We are giving an eye to copy-and-paste, which is a commonly performed operation when developers are writing source code [9], [10]. In existing differencing techniques, a change made by developer's copy-and-paste is visualized as a sequence of new code insertions. As a result, long edit scripts are generated for changes on which developer performed copy-and-paste operations.

In this research, we propose to consider copy-and-paste as a kind of editing actions in edit scripts. Introducing copy-and-paste to edit scripts contributes to not only making them shorter but also making them closer to developers' actual editing sequences. In other words, our technique generates edit scripts for more easily understanding code changes. Our technique is a lower-level representation of changes, but the authors consider that it can be used to generate higher-level abstracted changes. Higher-level changes are useful in various contexts of software development such as version control merging [11], [12]. We conduct an experiment on 14 open source software with an implemented tool of our technique. The followings are main findings of the experiment.

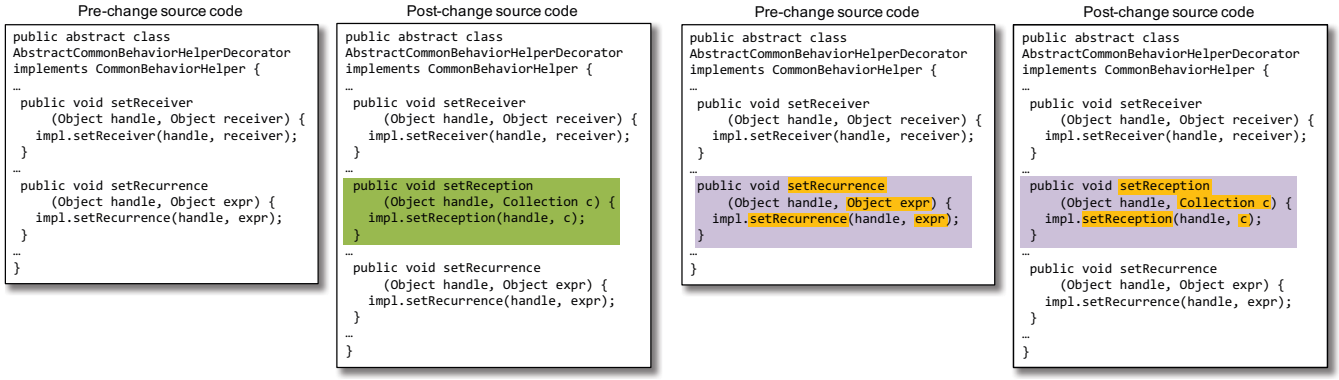
- Our technique generated shorter edit scripts for 18% changes than Falleri's technique [8]. For the remaining 82% changes, our technique generated the same edit scripts as Falleri's one.
- Our technique took longer time to generate edit scripts than Falleri's technique. However, for 96% changes, generation time was less than one-and-a-half times than Falleri's technique. For 96% changes, our technique generated edit scripts in two seconds.
- The visualization of our technique was more helpful than Falleri's technique for 12 research participants on all the ten change understanding tasks.

The reminder of this paper is organized as follows: Section II shows an actual example for which our technique generates a better edit script and then lists RQs to reveal in this research; Section III introduces some terminologies

[1]Abstract Syntax Tree

978-1-5386-2684-9/17/$31.00 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research



(a) GumTree's visualization

(b) Ours visualization

Fig. 1. An actual change where our technique generated a shorter edit script than Falleri's technique

for this research; Section IV explains edit script and our research motivation; in Section V, we propose a new AST edit script generation technique by considering copy-and-paste; Section VI shows our experimental results and then answers the RQs; Section VII dicsusses the results; Section VIII describes some threats in the experiment; lastly, we conclude this paper in Section IX.

II. A Change Example and Research Questions

Figure 1 shows an actual change for which our technique generates a different edit script from Falleri's one [8]. In this change, a method named *setReception* has been newly added. *setReception* has a similar structure to another method named *setRecurrence* while the used variables and the invoked methods in the two methods are different.

Figure 1(a) shows a visualized change with Falleri's technique. We can easily understand that green-colored *setReception* has been added by the change. Figure 1(b) shows a visualized change with our technique. The purple area in the pre-change code means a copied code fragment and the same color in the post-change code means its pasted code fragment. The yellow tokens in the purple areas are different tokens between the copied and pasted code fragments.

By using Falleri's technique, we can easily understand that *setReception* has been added. However, there is no information about the followings in the visualization.

- *setReception* is similar to *setRecurrence*.
- The developer might have copied and pasted *setRecurrence*, and then he/she might have updated some tokens inside the pasted code.

By using our technique, we can easily obtain the above information. The knowledge that *setReception* is similar to *setRecurrence* should be useful for program understanding. For example, if a developer is trying to understand *setReception* and he/she knows that *setReception* is similar to *setRecurrence* with which he/she is familiar, he/she should be able to easily understand *setReception*.

Falleri's edit script for this change includes 20 editing actions while our edit script includes only five. Our technique shortens the edit script for this change by 75%.

In this research, we try to answer the following research questions.

RQ1: how often and how much does our technique generate shorter edit scripts than Falleri's one?
RQ2: can our technique generate edit scripts at short times?
RQ3: are edit scripts of our technique more helpful to understand source code differences than Falleri's one?

III. Preliminaries

A. Abstract Syntax Tree

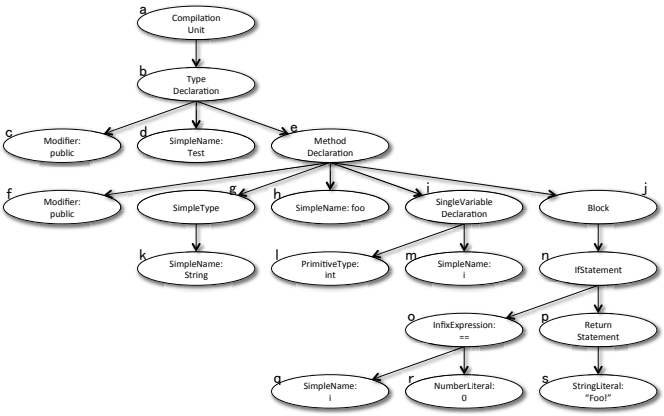
Abstract Syntax Tree (in short, AST) is a tree-structured representation of source code. Figure 2 shows toy source code and an AST generated from it. This AST has 19 nodes, each of which corresponds to a program element in the source code. Each node has a label for type and some nodes have values too. For example, in Figure 2(b), "NumberLiteral: 0" means that "*NumberLiteral*" is its node type and "*0*" is its value. If a node has child nodes, the child nodes represent more detailed information of the node. For example, the *Ifstatement* node whose ID is *n* has two child nodes, an *InfixExpresson* node and a *ReturnStatement* node. They represent the conditional expression and the inner statement of the if-statement.

B. Copy-and-Paste

When developers are writing source code, they often copy and paste code [10]. There is a study that 64% copy-and-paste is performed within a single source file [9]. Copied and pasted code become code clones. Kim et al. proposed *clone genealogy*, which approximates how programmers create, propagate, and evolve code clones by copying, pasting, and modifying code [13]. Li et al. pointed out that copy-and-pasted code can include bugs because developers occasionally forget to change identifier names after copy-and-pasted operations [14].

```
public class Test{
 public String foo(int i){
   if(i == 0) return "Foo!";
 }
}
```

(a) Source code



(b) AST

Fig. 2. An AST example

They developed a tool, CP-Miner, to find such copy-and-pasted related bugs and found several dozen of new bugs in open source operating systems. Jablonski et al. developed an Eclipse plug-in to track code generated by copy-and-paste operations [15]. The plug-in monitors developer's behavior on Eclipse to catch copy-and-paste operations.

C. Code Clone

A code clone (in short, clone) is a code fragment that is similar or identical to another code fragment in source code. Clones occur in source code for various reasons [16], [17]. Clones are classified as follows based on the degree of similarity to their correspondences.

- **TYPE-1** clone is completely identical code except white spaces, tabs, new-line characters, and comments.
- **TYPE-2** clone is similar code that includes token-level differences. For example, similar code having different variable names or literals are classified into TYPE-2.
- **TYPE-3** clone is similar code that includes larger differences than token-level. For example, if a program statement is inserted or deleted after a copy-and-paste operation, the pasted code becomes a TYPE-3 clone of its original code.

Various clone detection techniques have been proposed until now [17], [18]. Each detection technique has a different clone definition. Thus, different detection techniques find different clones from the same source code.

In this research, clones are similar subtrees in ASTs. Our TYPE-1 clones mean identical subtrees, which have the same structures and the same values in their nodes. Our TYPE-2 clones mean they have the same structures but include different values. Our TYPE-3 clones mean their structures are similar but not identical, including some extra nodes compared with their correspondences. Currently, our technique utilizes only TYPE-1 and TYPE-2 clones for copy-and-paste operations.

IV. AST Edit Script

An AST edit script is a sequence of editing actions to transform a given AST to another AST. In existing research, the following edit actions are considered [4], [5], [8].

- $insert(t, t_p, i, l, v)$ means inserting a new node to the AST. t is the inserted node. Its label is l. Its value is v. Its parent node is t_p. i means that t is the i-th child of t_p.
- $delete(t)$ means deleting an existing node from the AST. t is the deletion target.
- $update(t, v)$ means updating value of an existing node in the AST. t is the updating target node. v is a new value.
- $move(t, t_p, i)$ means moving a subtree to another place in the AST. t is the root node of the moving target subtree. t_p is the new parent node after t was moved. i means that t is the i-th child of t_p.

The length of an edit script is the number of editing actions included in it. Previous research reported that longer edit scripts require more effort to understand [8]. Consequently, shorter edit scripts are better from the viewpoint of understanding code changes.

An edit script shows how to transform an AST to another one. However, edit scripts do not necessarily reproduce the process of code changes that developers made. If we want to reproduce such an actual change process, we need to record developers' changes themselves [15], [19].

A. Previous Research on Edit Script Generation

There are many research studies that proposed techniques of edit script generations. Myers proposed an efficient algorithm to compare two strings (A and B) and then generate a shortest edit script that transforms A into B [3]. The algorithm requires $O(ND)$ time. N is the sum of the lengths of A and B. D is the size of the shortest edit script for A and B. Miller et al. proposed another algorithm for string comparison [2]. Their algorithm is faster than Unix diff in the case where two very similar strings are compared. However, if two completely different strings are compared, the performance of the algorithm is much worse than diff. Both Myers's and Miller's algorithms do not consider moving code.

Asaduzzaman et al. proposed a technqiue named LHDiff for better tracking of source code lines [1]. Firstly, LHDiff utilizes Unix diff to identify unchanged code lines in given two source files. Then, for the remaining code lines, the technique computes their context and content similarity. If it identifies similar code lines, they are regarded as moved code. In Asaduzzaman's experiment, the time of LHDiff execution was 20 $\sim$ 30 times longer than Unix diff.

Recently, tree edit script generations have been researched well. Pawlik et al. proposed RTED algorithm to compute a shortest tree edit script [7]. In the worst case, their algorithm requires $O(n^3)$ time where n is the number of tree nodes. However, RTED does not consider moving code. Their experiment showed that RTED algorithm worked more efficiently for any

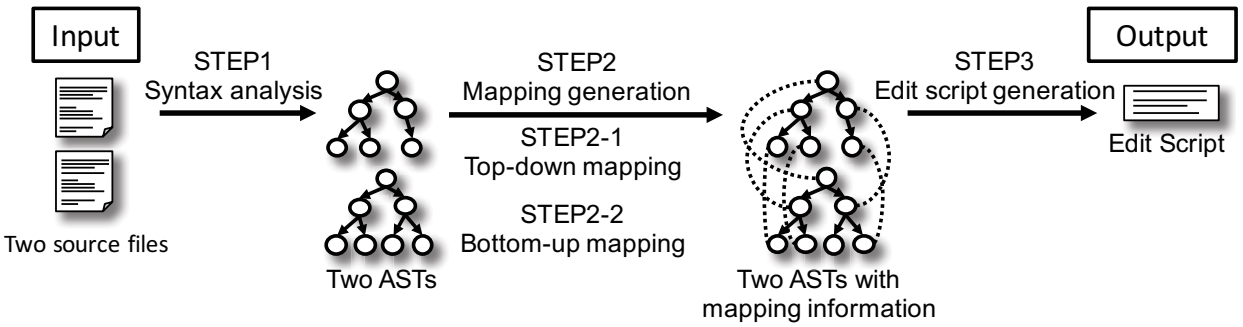


Fig. 3. An overview of GumTree

shapes of trees than other algorithms that generate a shortest edit script and do not consider moving code.

Chawathe et al. proposed an algorithm to generate an edit script from given two trees [4]. The remarkable feature of the algorithm is that the algorithm considers moving code. Generating a shortest edit script in consideration with moving code is an NP-hard problem. To finish edit script generation quickly, the algorithm often generates longer edit scripts. In the case where two similar trees are given to the algorithm, it always generates a shortest edit script. The algorithm requires $O(ne+e^2)$ time where n is the number of tree leaves and e is the weighted edit distance (typically, $e \ll n$).

Fluri et al. proposed an algorithm to compute tree differencing for source code change extraction [5]. This algorithm is a specialized version of Chawathe's algorithm [4] because Chawathe's algorithm is not for AST but for the general tree structure. Fluri et al. introduced four kinds of modifications on Chawathe's algorithm to extract more significant source code changes. Because of the modifications, Fluri's algorithm gets $O(log\ n^2)$ slower than Chawathe's algorithm. However, in the experiment, Fluri's algorithm approximated the minimum edit scripts 45% better than the Chawathe's algorithm.

Hashimoto et al. proposed a technique to generate edit scripts including code move [6]. Their technique utilizes Zhang's algorithm [20], which occasionally generates unfitting AST edit scripts because it is for general tree structures, not specialized for AST structures. Thus, Hashimoto et al. introduced some preprocessing and postprocessing to generate more appropriate AST edit scripts. In Hashimoto's technqiue, move operations are derived from pairs of deleted subtrees and added subtrees satisfying some conditions. Even in the worst case, the time complexity of Hashimoto's technique cannot be higher than $O(n^2)$ where n is the number of nodes in the compared trees.

There are some algorithms that generate edit scripts for XML documents [21], [22]. Their algorithms give shortening the computational time the utmost importance. Thus, edit scripts generated with them are not suited to be understood by the human.

```
public class Test{
 private String foo(int i){
   if(i == 0) return "Bar";
   else if(i == -1) return "Foo!";
 }
}
```

(a) Source Code After Change

```
insert(t1, n, 2, ReturnStatement, ε)
insert(t2, t1, 1, StringLiteral, Bar)
insert(t3, n, 3, IfStatement, ε)
insert(t4, t3, 1, InfixExpression, ==)
insert(t5, t4, 1, SimpleName, i)
insert(t6, t4, 2, PrefixExpression, -)
insert(t7, t6, 1, NumberLiteral, 1)
move(p, t3, 2)
update(c, private)
```

(b) Generated Edit Script

Fig. 4. An example of GumTree's edit script

B. GumTree

Falleri et al. proposed a technique, **GumTree**, which can generate edit scripts from larget ASTs in consideration with moving code at short times [8]. Figure 3 shows an overview of GumTree. If GumTree takes the source code of Figure 2(a) and 4(a) as its inputs, it outputs the edit script shown in Figure 4(b).

GumTree executes the following steps for given two source files, and then it outputs an edit script.

- **STEP1 (Syntax analysis)** performs syntax analysis for given source files to generate an AST for each of them.
- **STEP2 (Mapping generation)** generates mappings of subtrees between the two ASTs by executing STEP2-1 and STEP2-2.
  - **STEP2-1 (Top-down mapping)** searches identical subtrees between the two ASTs. The search starts at the root nodes of the ASTs.
  - **STEP2-2 (Bottom-up mapping)** searches similar subtrees from other than the identical subtrees found in STEP2-1. This search begins with each leaf node of the ASTs.
- **STEP3 (Edit script generation)** generates an edit script based on the mappings generated in STEP2-1 and STEP2-

2. Chawathe's algorithm [4] is used to generate edit scripts.

C. Research Motivation

Developers occasionally copy existing code fragments and paste them in other places of the source code when they are writing code [10]. However, GumTree does not consider copy-and-paste operations. In edit scripts of GumTree and other differencing techniques, a copy-and-paste is represented with a sequence of *insert* actions.

Figure 5 shows a code change example where a new method is added to the source code shown in Figure 2(a). The added method *newFoo* is quite similar to the existing method *foo*. Figure 5(b) shows an AST of the post-change source code. For this change, GumTree outputs an edit script shown in Figure 5(c). This edit script means that 15 new nodes were added to the AST by the change.

However, the subtree of the added method has the same structure as the subtree of the existing method. In this case, by considering copy-and-paste, a simpler edit script can be generated. Figure 5(d) shows an edit script generated by our technique. In this edit script, firstly the subtree of the existing method is copied and pasted, then two nodes in the pasted subtree are updated.

V. Proposed Technique

Herein, we explain our technique. An implemented tool of our technique for Java is publicly available [2]

A. Outline

As described in Subsection IV-C, by representing an added subtree with a copy-and-paste action rather than a sequence of insert actions, simpler edit scripts can be generated. In our technique, a new editing action representing copy-and-paste is introduced in addition to the conventional editing actions (*insert*, *delete*, *update*, and *move*). In the case of Figure 5, the length of the edit script reduces by one-fifth. Our edit script is not only shorter but also easier-to-understand because a combination of copy-and-paste and update actions implies that similar code has been added by a given change.

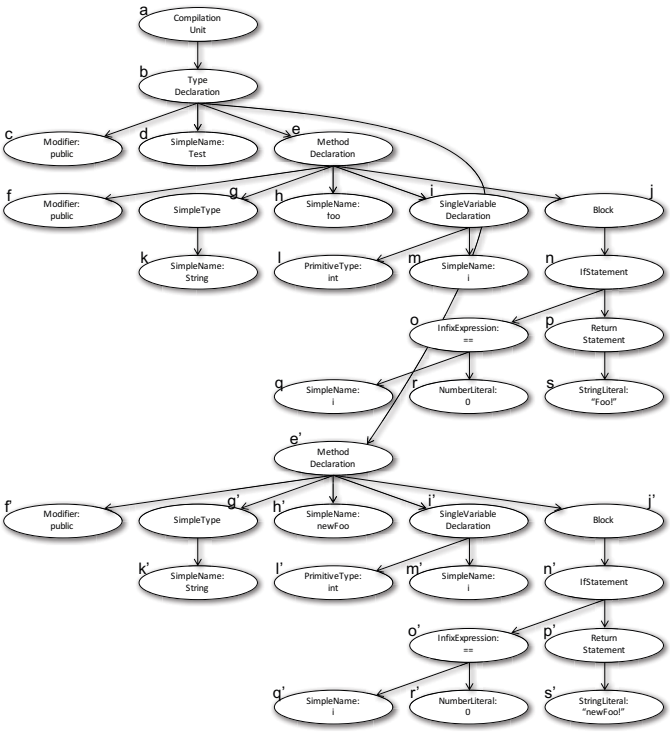
Our technique is designed by extending GumTree [8]. Figure 6 shows an overview of our technique. Our technique takes two source files, and it outputs an edit script. Our technique includes three steps as well as GumTree. Our technique considers five editing actions, *insert*, *delete*, *update*, *move*, and *c&p*. The former four actions are the same as GumTree's ones which are described in Section IV. The following is the definition of the new action.

- $c\&p(t, t_p, i)$ means copying an existing subtree and pasting it to another place in the AST. t is the root of the subtree of the copying target. t_p is the parent node of the pasted place. i means that t is pasted as the i-th child of t_p.

[2]http://sdl.ist.osaka-u.ac.jp/~higo/ase2017/

```
public class Test{
 public String foo(int i){
   if(i == 0) return "Foo!";
 }
 public String newFoo(int i){
   if(i == 0) return "newFoo!";
 }
}
```

(a) Post-change code



(b) Post-change AST

```
insert(e', b,  4, MethodDeclaration, ε)
insert(f', e', 1, Modifier, public)
insert(g', e', 2, SimpleType, String)
insert(k', g', 1, SimpleName, String)
insert(h', e', 3, SimpleName, newFoo)
insert(i', e', 4, SingleVariableDeclaration, ε)
insert(l', i', 1, PrimitiveType, int)
insert(m', i', 2, SimpleName, i)
insert(j', e', 5, Block, ε)
insert(n', j', 1, IfStatement, ε)
insert(o', n', 1, InfixExpression, ==)
insert(q', o', 1, SimpleName, ε)
insert(r', o', 2, NumberLiteral, 0)
insert(p', n', 2, ReturnStatement, ε)
insert(s', p', 1, StringLiteral, newFoo!)
```

(c) Edit script by GumTree

```
c&p(e, b, 4)
update(h', newFoo)
update(s', "newFoo!")
```

(d) Edit script by our technique

Fig. 5. A change example where a new method was added

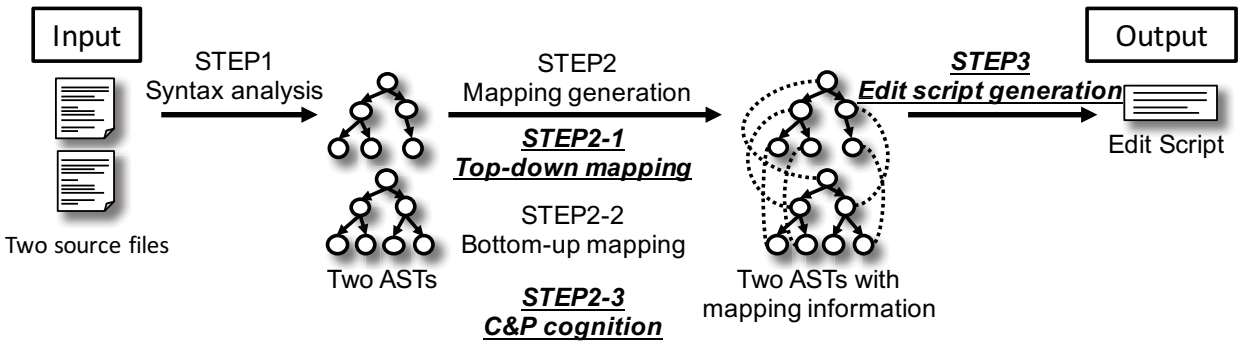


Fig. 6. An overview of our technique

B. Procedure

Our technique is an extended version of GumTree. In Figure 6, the steps that have been added or changed in our technique are emphasized. The followings are short descriptions for each of the steps.

- **STEP2-1 (Top-down mapping)** searches identical subtrees between the two ASTs. This step also finds candidates for copy-and-pasted subtrees.
- **STEP2-3 (C&P cognition)** checks whether each of the copy-and-pasted candidates has been mapped in STEP2-2 or not. If not, the candidate is regarded as generated by copy-and-paste.
- **STEP3 (Edit script generation)** generates an edit script with the two ASTs generated in STEP1 and the mapping information (including the copy-and-paste information).

The remainder of this section explains each of the above steps in detail.

C. STEP2-1 (Top-Down Mapping)

In STEP2-1, our technique finds candidates of subtrees added by copy-and-paste operations in addition to the processings of original GumTree. STEP2-1 of our technique consists of the following processings. The processings (1) and (2) are the same as original GumTree. The processing (3) has been newly added in our technique.

1) Finding similar subtrees between the two ASTs. In this explanation, we assume that n subtrees in the pre-change AST are similar to m subtrees of the post-change AST. Those subtrees can be represented by a bipartite graph where a set of n nodes and another set of m nodes exist. An edge exists between each pair of two nodes whose similarity is higher than a given threshold.
2) Making mappings between the n nodes and the m nodes with the following procedure. Firstly, finding a pair of nodes that has the highest similarity between the n nodes and the m nodes. The found pair is recorded and the two nodes forming the pair are removed from the bipartite graph. Secondly, finding a pair of nodes that has the highest similarity between the $n-1$ nodes and the $m-1$ nodes. This processing is repeated until all edges in the bipartite graph disappear.
3) When the processing (2) has finished, the remaining nodes in the bipartite graph mean that their subtrees have not been mapped to any other subtrees. In other words, their subtrees are similar to other subtrees, but the other subtrees have more similar subtrees. In our technique, non-mapped subtrees in the post-change AST are treated as candidates for copy-and-paste from the most-similar subtrees in the pre-change AST.

D. STEP2-3 (C&P Cognition)

In this step, our technique determines which candidates are copy-and-paste operations. Our technique traverses the post-change AST to find nodes satisfying both the following conditions:

- found as candidates for copy-and-paste operation in STEP2-1, and
- not mapped to any nodes in the pre-change AST when STEP2-2 has finished.

If our technique finds a node that satisfies both the conditions, it regards a pair of the candidate and its most-similar subtrees as a copy-and-paste mapping.

In this step, we use a heuristic. In our technique, subtrees' similarities are calculated based on their structure and node labels. Node values are ignored. Thus, many false positives are found. For example, all variable declaration statements are regarded as identical subtrees even if their variable types and variable names are different. To reduce such false positives, if all node values in a subtree are different from ones of another subtree, the two subtrees are not regarded as a copy-and-paste operation.

E. STEP3 Edit Script Generation

Chawathe's algorithm is used in STEP3 to identify *insert*, *delete*, *update*, and *move* as well as original GumTree. Our technique extends Chawathe's algorithm to identify *c&p* too. In STEP3, the two ASTs are traversed once and an edit script is generated.

1) Identifying *insert*, *update*, *move*, and *c&p* actions by traversing the post-change AST.
2) Identifying *delete* actions by traversing the pre-change AST.

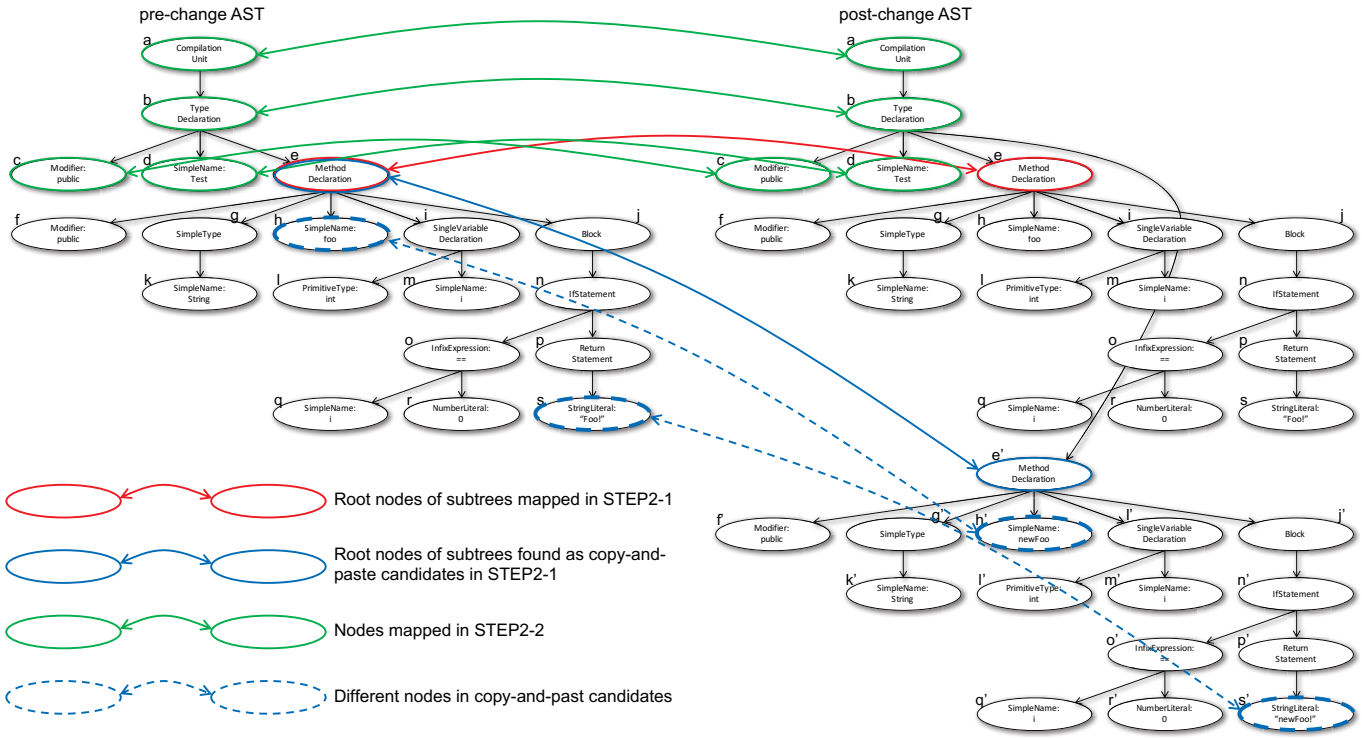


Fig. 7. An example of mappings (mappings between Figure 2(b) and Figure 5(b))

In the processing (1), if our technique finds a node in the copy-and-paste mappings, a *c&p* action is added to the edit script.

F. An Example of Edit Script Generation

Figure 7 shows an example of mappings between pre-change and post-change ASTs. In this figure, the pre-change AST is the same as Figure 2(b), and the post-change AST is the same as Figure 5(b).

In STEP2-1 (Top-down mapping), subtree e in the pre-change AST gets mapped with subtree e in the post-change AST. Besides, subtree e in the pre-change AST is very similar to e' in the post-change AST, so that they become a candidate for copy-and-paste operation.

In STEP2-2 (Bottom-up mapping), firstly container mappings are generated. A container mapping means, if most nodes under a subtree have been mapped, the root node of the subtree also gets mapped. In the process of container mapping generation, node b in the pre-change AST gets mapped with node b in the post-change AST. Then, recovery mappings are generated. Recovery mappings mean mapping nodes by traversing from the nodes of the container mappings to their leaves. In this example, nodes c and d get mapped in the process of recovery mapping generation.

In STEP2-3 (C&P cognition), each copy-and-paste candidate found in STEP2-1 is checked whether it has been mapped in STEP2-2 or not. If not, the candidate is regarded as copy-and-paste operation. In this example, subtree e in the pre-change AST and e' of the post-change AST found as a copy-and-paste candidate in STEP2-1, and e' is not mapped in STEP2-2. Consequently, they are regarded as a copy-and-paste operation in STEP2-3.

In STEP3, an edit script is generated with the mapping and copy-and-paste information as follows.

- For each pair of moved subtrees, a *move* action is added to the edit script. In this example, there are no moved subtrees.
- For each pair of nodes that includes different values, an *update* action is added to the edit script. Node pairs under a pair of copy-and-paste are not targets of this processing. Thus, no *update* action is added to the edit script in the example.
- If the pre-change AST includes nodes that are not mapped to any nodes in the post-change AST, *delete* actions for them are added to the edit script. In this example, there is no node for *delete* actions.
- If the post-change AST includes nodes that are neither mapped nor copy-and-paste, *insert* actions for them are added to the edit script. In this example, there is no node for *insert* actions.
- For each subtree of copy-and-paste in the post-change AST, a *c&p* action is added to the edit script. If there are nodes under the subtree that have different values from their counterparts, *update* actions for them are added to the edit script. In this example, action `c&p(e,b,4)` is added and then two more actions `update(h', newFoo)` and `update(s',"newFoo!")` are also added.

As a result, the edit script shown in Figure 5(d) is generated by our technique.

VI. Evaluation

As described in Section IV-A, there are many techniques to generate edit scripts. Falleri's experiment showed that GumTree's edit scripts were easier to understand for the human than other techniques [8]. Hence, we compare our technique with GumTree and then answer the RQs listed in Section II.

A. Preparation

We use the following values as the thresholds of GumTree and our technique. Those values are the same as ones that were used in Falleri's experiment [8].

- The minimum subtree height for top-down mappings is 2.
- The minimum similarity for bottom-up mappings is 0.5.
- Only subtrees having 100 or fewer nodes are targets of bottom-up mappings.

Our experimental targets are 14 software systems that are included in CVS-Vintage dataset [23]. This dataset includes 42,250 source files and 352,182 revisions in total. This dataset was used in Falleri's experiment [8] too.

B. Procedure for RQ1 and RQ2

We run GumTree and our technique for the same changes and then compared generated edit scripts[3]. In the remainder of this section, we call a pair of two consecutive revisions of a source file *a change*. Thus, the number of changed source files is the same as the number of changes in this experiment. Changes for the two tools were extracted from the 14 projects. For each project, 1,000 changes were extracted. If a project included less than 1,000 changes, all the changes were extracted. The target changes were extracted in the following way, which is the same as Fellari's experiment.

1) We identified revisions in which at least a source file is committed, and then we obtained a list of the committed source files for each of the identified revisions.
2) We retrieved the pre-change revision for each of the files in the lists.
3) We stored each pair of the pre-change and post-change texts of the files as a change.
4) We removed changes where only comments and/or formatting are modified because empty edit scripts are generated from such changes.
5) We selected 1,000 changes randomly from the remaining ones.

We obtained 13,699 changes in total. All the obtained changes were given to GumTree and our technique to generate edit scripts. The execution time of the tools was measured.

[3]GumTree and our technique were executed on a personal workstation equipped with a 2.40GHz 6-core CPU and 32GB DDR4 memory.

TABLE I
EDIT SCRIPT LENGTH OF OUR TECHNIQUE AND GUMTREE FOR THE CHANGES FOR WHICH OUR TECHNIQUE GENERATED DIFFERENT EDIT SCRIPTS FROM GUMTREE

Project	Maximum value		Median		Minimum value	
	GT	Ours	GT	Ours	GT	Ours
argouml	3,049	2,708	69	49	4	2
carol	1,586	1,581	116	107	7	3
columba	892	872	47	39	5	1
dnsjava	1,632	1,544	75	71	4	2
jboss	2,876	2,457	76	62	5	2
jedit	2,858	2,853	79	67	4	2
jhotdraw	1,691	1,686	72	64	4	2
junit	751	728	75	64	7	2
log4j	3,029	2,827	75	60	4	2
jdtcore	13,269	12,628	84	65	4	2
workbench	3,651	3,351	72	55	4	2
scarab	3,016	2,928	83	70	5	2
struts	1,486	1,313	59	43	5	1
tomcat	2,152	2,064	54	43	4	2

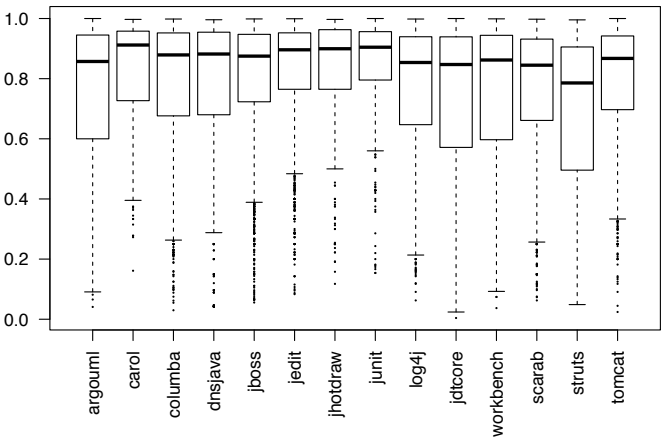


Fig. 8. Edit script length ratio of our technique to GumTree

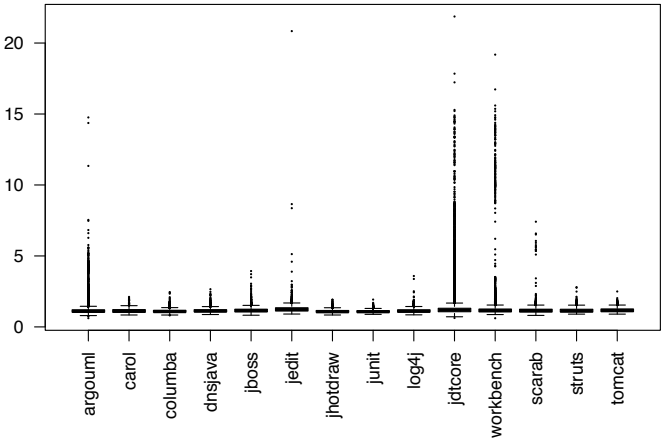


Fig. 9. Execution time ratio of our technique to GumTree

C. Results for RQ1 and RQ2

For 18% changes, our technique generated shorter edit scripts than GumTree. For the remaining 82% changes, our technique generated the same edit scripts as GumTree. There was no change for which our technique generated a longer edit script than GumTree.

Table I shows the length of the edit scripts for the 18% changes. Figure 8 shows the length ratio of our technique to GumTree for the 18% changes. For most of the projects, the median values are between 0.8 and 0.9. More concretely, for 58.5% of the changes in the graph, our technique shortened edit scripts by 10% or more.

Figure 9 shows the execution time ratio of our technique to GumTree. The execution time of our technique tends to longer than GumTree. For all the changes, we confirmed that there was a significant difference in execution time between our technique and GumTree by using Wilcoxon signed-rank test with $\alpha = 0.05$. However, the difference of execution time is not so large. For 96% changes, our technique took one-and-a-half times or less than GumTree. For 75% and 96% changes, our technique took less than 1 second and 2 seconds, respectively.

Our answer to RQ1 is that our technique generated shorter edit scripts for 18% changes and for 58.5% of those changes our technique shortened 10% or more.

Our answer to RQ2 is that our technique can generate edit scripts at short times. Our technique took less than 2 seconds for 96% of all target changes.

D. Procedure for RQ3

Twelve research participants took part in the experiment for RQ3. The participants include one professor, nine graduate students, and two undergraduate students. The professor and all the graduate students had experiences in the research of software engineering. They had at least 1-year experiences of Java programming in their research. The undergraduate students had finished a half-year Java exercise in their course. All the participants are not the authors of this paper.

In this experiment, the authors prepared ten changes (hereafter, we call them *tasks*), all of which were selected from the 13,699 changes. All the ten tasks satisfy both the following conditions.

- GumTree and our technique generate different edit scripts for the tasks.
- The tasks do not include huge code changes such as changing all over the source files.

This experiment included two phases. In the first phase, the research participants were divided into two groups. The participants in the first group used our technique to understand the changes for the odd-numbered tasks and they used GumTree for the even-numbered tasks. The participants in the second group used GumTree and our technique in an opposite way. Each of the participants wrote down what he/she understood on the tasks after they had finished understanding the tasks. To keep participants' concentration, we set one hour as the time limit for understanding the tasks. Thus, there were some participants who were not able to finish some tasks.

In the second phase, for each of the tasks, each participant compared visualization of our technique with GumTree and then chose from the following five options.

1) The visualization of GumTree is definitely more helpful to understand the tasks.
2) The visualization of GumTree seems a little more helpful to understand the tasks.
3) No differences in the visualization of GumTree and our technique from the viewpoint of understanding the tasks.
4) The visualization of our technique seems a little more helpful to understand the tasks.
5) The visualization of our technique is definitely more helpful to understand the tasks.

Fig. 10. Understanding time for the tasks

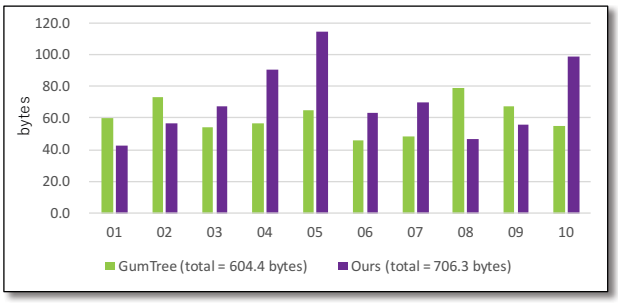


Fig. 11. Description length for the tasks

E. Evaluation Measure for RQ3

In this experiment, we used the following evaluation indicators to compare our technique with GumTree:

- the time required to understand the tasks,
- the degree of understanding the tasks, and
- participants' qualitative comparison.

The participants timed themselves for the time required to understand the tasks. Before this experiment, we told the participants not to count the time required to write down what they understood on the tasks in the time required to understand the tasks.

We considered that, if a participant understood a task well, he/she would write down what she/he understood in detail. Thus, we used the length of participants' descriptions as the degree of understanding the tasks.

We used the questionnaire described in the last paragraph of Subsection VI-D for the qualitative comparison.

F. Results for RQ3

Figure 10 shows the average understanding time of the research participants for each of the tasks. To reduce influences of the outliers, we eliminated the minimum and maximum values in calculating average time. The total time of our technique was 2,187 seconds and GumTree is 2,220 seconds, respectively. To investigate whether there was a significant difference in understanding time between our technique and GumTree, we firstly checked the presence of normality in the time data with Shapiro-Wilk test. After we confirmed the presence of normality, we applied Paired-T test to the

TABLE II
ANSWERS OF THE RESEARCH PARTICIPANTS FOR QUALITATIVE COMPARISONS

		Research participants													
		A	B	C	D	E	F	G	H	I	J	K	L	Ave.	Med.
Tasks	01	1	3	4	2	2	5	4	5	5	5	2	2	3.33	3.5
	02	1	4	5	4	4	5	3	5	5	5	2	4	3.92	4.0
	03	1	2	5	2	5	5	5	5	5	4	5	1	3.75	5.0
	04	1	4	4	2	4	5	5	5	3	5	5	4	3.92	4.0
	05	1	4	5	1	5	5	4	5	5	5	2	3	3.75	4.5
	06	1	2	1	2	3	5	3	5	4	5	4	4	3.25	3.5
	07	1	4	3	5	–	5	5	5	5	4	4	5	4.18	5.0
	08	1	4	3	2	–	4	3	5	4	4	2	3	3.18	3.0
	09	1	2	3	3	–	5	3	–	4	4	–	–	3.13	3.0
	10	1	5	3	3	–	5	3	–	5	5	–	–	3.75	4.0
	Ave.	1.00	3.40	3.60	2.60	3.83	4.90	3.80	5.00	4.50	4.60	3.25	3.25		
	Med.	1.0	4.0	3.5	2.0	4.0	5.0	3.5	5.0	5.0	5.0	3.0	3.5		

data. As a result, there was no significant difference in the understanding time between our technique and GumTree.

Figure 11 shows the average length of participants' descriptions for each of the tasks. We eliminated the minimum and maximum values in calculating average length, too. The total length of our technique was 706.3 bytes, and GumTree was 604.4 bytes, respectively. To investigate the presence of a significant difference in the description length of our technique and GumTree, we firstly checked the presence of the normality in the description length data with Shapiro-Wilk test. After we confirmed that there was no normality in the data, we applied Wilcoxon signed-rank test to the data. As a result, there was no significant difference in the description length between our technique and GumTree.

Table II shows the participants' answers for the questionnaire. For all the tasks, the average and median values were more than 3[4]. In other words, the research participants felt that our technique was more helpful than GumTree for all the tasks. On the other hand, there were two participants whose average and median values were less than 3. Research participants A and D considered that GumTree was better than our technique for the tasks. Consequently, we conclude that our technique provides better visualization than GumTree. However, there were some developers who preferred GumTree's visualization than our technique.

Our answer to RQ3 is that the research participants tended to prefer change visualization based on our technique than Falleri's one for all their tasks.

VII. DISCUSSION

In the current implementation, identical subtrees (same structure and same values) and structurally-identical subtrees (same structure but different values) are regarded as copy-and-paste. The former is a TYPE-1 clone and the latter is a TYPE-2 clone, respectively. In the experiment, for 82% changes, our technique generated the same edit scripts as GumTree. Those edit scripts did not include *c&p* actions at all. If we use an AST-based TYPE-3 clone detection techniques such as CloneDR [16] or Deckard [24], our technique can be extended to regard TYPE-3 clones as copy-and-paste. If we do such an extension, more edit scripts will include *c&p* actions.

We used value "2" as the minimum height of subtrees for copy-and-paste. If we use a greater value, less edit scripts include *c&p* actions.

[4]In the questionnaire, 1. $\sim$ 5. are ordinal scale, not ratio scale. Thus, the average values in this experiment should be used only as a guide.

VIII. THREATS TO VALIDITY

In the experiment, the target programming language was only Java. Our technique's and GumTree's algorithms are not specialized for Java. However, currently, we do not know whether our technique works well for other programming languages.

In the experiment, we used the same experimental targets and the same threshold values as Falleri's experiment. However, if we use different projects or different threshold values, we may obtain different experimental results. More experiments are required with more projects and more different threshold values to evaluate our technique more solidly.

IX. CONCLUSION

In this paper, we proposed to introduce copy-and-paste operation to AST edit script to promote change understandability. In the process of designing our technique, we were greatly affected by an existing technique GumTree and our technique is its extended version. We conducted an experiment to compare our technique with GumTree. As a result, our technique generated shorter edit scripts than GumTree for 18% changes. For the remaining 82% changes, our technique generated the same edit scripts as GumTree. We also confirmed that the execution time of our technique tends to be longer than GumTree. However, for 96% changes, the execution time of our technique was less than 2 seconds. We conducted another experiment with 12 research participants and confirmed that our technique provided more helpful visualization than GumTree for all the ten change understanding tasks.

In the future, we are going to apply our technique for more projects. Extending our technique to regard TYPE-3 clones as copy-and-paste is another future work.

ACKNOWLEDGMENT

This work was supported by MEXT/JSPS KAKENHI 25220003.

REFERENCES

[1] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. D. Penta, "LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines," in *Proc. of the 2013 IEEE International Conference on Software Maintenance*, 2013, pp. 230–239.

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

[3] E. W. Myers, "An o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[4] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change Detection in Hierarchically Structured Information," in *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 493–504.

[5] 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, Nov. 2007.

[6] M. Hashimoto and A. Mori, "Diff/TS: A Tool for Fine-Grained Structural Change Analysis," in *Proc. of the 2008 15th Working Conference on Reverse Engineering*, 2008, pp. 279–288.

[7] 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.

[8] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 313–324.

[9] T. M. Ahmed, W. Shang, and A. E. Hassan, "An empirical study of the copy and paste behavior during development," in *Proc. of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 99–110.

[10] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," in *Proc. of the 2004 International Symposium on Empirical Software Engineering*, 2004, pp. 83–92.

[11] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen, "Effective Software Merging in the Presence of Object-Oriented Refactorings," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 321–335, 2008.

[12] D. Dig, R. Johnson, D. Marinov, B. Bailey, and D. Batory, "COPE: Vision for a Change-oriented Programming Environment," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 773–776.

[13] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 187–196.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[15] P. Jablonski and D. Hou, "CReN: A Tool for Tracking Copy-and-paste Code Clones and Renaming Identifiers Consistently in the IDE," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, 2007, pp. 16–20.

[16] I. Baxter, A. Yahin, M. A. L. Moura, and L. Bier, "Clone detection using abstract syntax trees," *Proc. of the 14th International Conference on Software Maintenance*, pp. 368–377, Mar. 1998.

[17] D. Rattan, R. Bhatia, and M. Singh, "Software Clone Detection: A Systematic Review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[18] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions Software Engineering*, vol. 31, no. 10, pp. 804–818, Oct. 2007.

[19] T. Omori and K. Maruyama, "An Editing-operation Replayer with Highlights Supporting Investigation of Program Modifications," in *Proc. of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, 2011, pp. 101–105.

[20] K. Zhang and D. Shasha, "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems," *SIAM Journal of Computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[21] R. Al-Ekram, A. Adma, and O. Baysal, "diffx: An algorithm to detect changes in multi-version xml documents," in *Proc. of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, 2005, pp. 1–11.

[22] G. Cobena, S. Abiteboul, and A. Marian, "Detecting changes in xml documents," in *Inproceedings of the 18th International Conference on Data Engineering*, 2002, pp. 41–52.

[23] M. Monperrus and M. Martinez, "Cvs-vintage: A dataset of 14 cvs repositories of java software," INRIA, Tech. Rep. hal-00769121, 2012.

[24] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 96–105.