Tree-based Mining of Fine-grained Code Changes to Detect Unknown Change Patterns

Yoshiki Higo*, Junnosuke Matsumoto*, and Shinji Kusumoto*
*Graduate School of Information Science and Technology, Osaka University, Japan
{higo, j-matumt, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—In software development, source code is repeatedly changed due to various reasons. Similar code changes are called change patterns. Identifying change patterns is useful to support software development in a variety of ways. For example, change patterns can be used to collect ingredients for code completion or automated program repair. Many research studies have proposed various techniques that detect change patterns. For example, Negara et al. proposed a technique that derives change patterns from the edit scripts. Negara's technique can detect fine-grained change patterns, but we consider that there is room to improve their technique. We found that Negara's technique occasionally generates change patterns from structurally-different changes, and we also uncovered that the reason why such change patterns are generated is that their technique performs text comparisons in matching changes. In this study, we propose a new change mining technique to detect change patterns only from structurally-identical changes by taking into account the structure of the abstract syntax trees. We implemented the proposed technique as a tool, TC2P, and we compared it with Negara's technique. As a result, we confirmed that TC2P was not only able to detect change patterns more adequately than the prior technique but also to detect change patterns that were not detected by the prior technique.**

***Index Terms*—Mining code change pattern, Repository mining, Edit script, Code change pattern**

I. Introduction

In software development, source code is repeatedly changed due to various reasons such as adding new functions, fixing exposed bugs, and improving code quality [1], [2], [3], [4]. Developers occasionally make similar changes to the source code. Change pattern is a term that means a set of changes similar to one other. The change pattern information is useful for developers and practitioners. The followings are typical situations where change patterns are useful.

- Developers in an IDE[1] can obtain candidates for code completion from change patterns instead of writing whole code by themselves [5], [6], [7].
- Library users can be alerted to misuse of libraries from change patterns of misuse corrections [8].
- Researchers of automated program repair can use change patterns as candidate modifications for program repair [9], [10], [11].

Many studies have been conducted to find change patterns. Most techniques to detect change patterns express code changes as edit scripts, which are sequences of edit operations, and identify frequently occurring edit operations in the edit scripts. However, some of those techniques are specialized to specific objectives, and they have limitations. For example, the techniques proposed in literature [6], [12] are specific to detect change patterns of method invocations. Coming [13] is a technique to identify change instances that match with given change patterns in Git repositories. This technique is not suited to detecting unknown change patterns.

Negara et al. proposed a general-purpose technique to detect change patterns [14]. Their technique derives change patterns from the edit scripts each of which are calculated from two versions of abstract syntax trees (in short, ASTs). Negara's technique can detect fine-grained change patterns, but we consider that there is room to improve their technique. We re-implemented Negara's technique as a tool and applied it to some repositories of open source projects[2]. As a result, we found that structurally-different changes are coincidentally consolidated into a change pattern in Negara's technique. We examined such change patterns and concluded that the reason why such change patterns were derived is the lack of considering locations of changed program elements in ASTs. In Negara's technique, texts of edit operations in changes are simply compared.

In this study, we propose a new change mining technique to detect change patterns only from structurally-identical changes by taking into account the AST structures. There are two key points in the proposed technique.

- The first point is constructing a *unified tree* from each commit in a Git repository. A *unified tree* is a tree structure that the ASTs before a given commit and the ones after the commit are combined.
- The second point is utilizing frequent tree pattern mining technique [15] on the *unified trees* instead of performing text comparisons on edit scripts.

Each change pattern detected by our technique consists of changes that transform the AST structures in the same way.

We have implemented the proposed technique (tree-based mining technique) as TC2P and compared it with a text-based mining technique [14]. The comparison results show that not only was TC2P able to prevent from detecting change patterns from structurally-different changes, but it was also able to detect the change patterns that were not detected by the text-based technique.

The remainder of this paper is organized as follows. In Section II, we explain our research motivation. In Section III, we explain some techniques used in this study. Then, we

[1]Integrated Development Environment

[2]We confirmed that Negara et al. did not publish their tool at 01/Feb/2020. Thus, we re-implemented a tool based on the descriptions in literature [14]. The re-implemented tool was also used in the comparisons in Section V.

introduce a new technique to detect tree-based change patterns in Section IV. We describe how we evaluated the proposed technique and the results in Section V. In Section VI, we describe prior studies related to this research. Finally, we conclude this paper in Section VII.

II. Research Motivation

Negara et al. proposed a technique to detect fine-grained unknown change patterns by mining edit operations included in edit scripts [14]. We consider that Negara's technique has an issue in terms of adequateness in detecting change patterns. Their technique occasionally consolidates different code changes into the same change pattern. We explain this issue with Figure 1. There are two changes in this figure. Each change inserts an if-statement, an expression-statement, and a return-statement while their relative positions are different. Thus, we consider that it is inadequate to make a common change pattern from the two changes. However, Negara's technique regards that those two changes form a common change pattern. In their technique, both changes are represented by the following edit operations.

- Inserting an element labeled as if-statement.
- Inserting an element labeled as expression-statement.
- Inserting an element labeled as return-statement.

Negara's technique identifies common edit operations by comparing edit actions (e.g., insertion, deletion) and target labels (e.g., if-statement, return-statement). Consequently, Negara's technique regards that those two changes include exactly the same edit operations, so that it makes a common change pattern from the changes. This fact has a negative impact on understanding change patterns because the change pattern can be interpreted as both changes. If a user wants to understand what kinds of changes are expressed by a given change pattern, he/she needs to see actual change instances included in the change pattern because there is no information on relative positions of program elements in the change pattern.

To overcome this issue, we propose a new technique to detect fine-grained change patterns from commit history. Our technique considers relative positions of program elements in addition to compare edit operations and labels. In the case of Figure 1, our technique does not make a common change pattern for the two changes. Moreover, change patterns derived by our technique can be understood by users without checking their actual change instances.

III. Preliminaries

In this section, we explain some terms and techniques that we use in this research. If you are eager to see our proposed technique, please skip this section and come back here if you meet unfamiliar terms/techniques in Section IV.

A. Abstract Syntax Tree

An abstract syntax tree (in short, AST) is a tree structure that represents source code. An AST is constructed for each source file in a project. Each node of an AST is composed of the following five elements.

ID. Each node includes a unique identifier in the AST.

```
  …
  void method(String name){

+   if(null == name){
+     this.initialized = false;
+     return;
+   }

    this.name = name;
  …
```

(a) Change A

```
    …
    String name = engineer.getName();

+   if(null == name){
+     return;
+   }
+   this.name = name;

    teamMembers.put(name, engineer);
    …
```

(b) Change B

Fig. 1. Motivating Example

Reference to parent. Each node has a reference to its parent node on the tree structure. The root node is the only exception because it does not have a parent node.

Reference to child. Each node has a reference to each of its child nodes on the tree structure. Leaf nodes do not have this reference because they do not have child nodes.

Label. Each node has a label that represents its grammatical type such as if-statement or return-statement.

Value. Some nodes have values that represent other information than the label. For example, nodes for variables include their variable names as their values.

B. GumTree

GumTree is a technique that detects differences between given two ASTs [16]. GumTree generates an edit script including a sequence of edit operations that converts one of the ASTs to the other. An edit operation consists of an edit action and its target nodes. GumTree supports the following four types of edit actions.

$\boldsymbol{insert(t, t_p, i, l, v)}$ 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.

$\boldsymbol{delete(t)}$ means deleting an existing node from the AST. t is the deletion target.

$\boldsymbol{update(t, v)}$ means updating the value of an existing node in the AST. t is the target node for updating. v is a new value.

$\boldsymbol{move(t, t_p, i)}$ 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 is moved. i means that t is the i-th child of t_p.

The number of editing operations included in an edit script is called the length of the edit script in this study. The processing of GumTree consists of the following two parts:

1) matching nodes between given two ASTs, and
2) generating an edit script based on the matching results.

The processing of matching nodes consists of two phases: *top-down* and *bottom-up*. First, in the *top-down* phase, GumTree traverses the two ASTs from their roots and maps subtrees that have exactly the same structures to each other in the two ASTs. In the *bottom-up* phase, GumTree traverses the two ASTs from the root nodes of the subtrees that have been mapped in the *top-down* phase to the root nodes of the two ASTs. In the *bottom-up* traverse, if GumTree finds subtrees that are similar to each other, it maps them. GumTree uses a value between 0 and 1 as a threshold to determine whether given two subtrees are similar or not. Subtree similarities are calculated with the Jaccard index of their nodes.

Then, GumTree computes an edit script by using the matching results. Based on the edit script, the nodes in the two ASTs are exclusively classified into the following three categories:

- nodes existing only in the before-change AST,
- nodes existing only in the after-change AST, and
- nodes existing in both ASTs.

For the nodes existing only in the before-change AST, GumTree regards that the change has deleted them. In the same way, for the nodes existing only in the after-change AST, GumTree regards that the change has inserted them. For the nodes existing in both ASTs, if their parent nodes are different between the before-change and after-change ASTs, GumTree regards that they have been moved. If values of nodes are different between the before-change and after-change ASTs, the nodes are regarded as updated. In this way, GumTree computes an edit script by using the matching results. The computation of the differences between the two ASTs has already been well optimized [17]. GumTree uses this technique.

GumTree has been used in a variety of studies. For example, it has been used for recommending APIs [6], analyzing Maven build files [18], repairing bugs automatically [11], [19], [20], and finding patterns in JavaScript bugs [21].

C. FREQT

FREQT is an algorithm to find frequent tree patterns from a labeled ordered tree [15]. A tree pattern is a structure common to subtrees in a given tree. By applying FREQT to *unified trees* (explained in Section IV-A), we can obtain change patterns that have been generated only from changes that transform the AST structures in the same way.

An ordered tree is a tree structure in which the child nodes of each node have an ordered relationship. A labeled ordered tree is an ordered tree in which each node has a label $l \in \mathcal{L}$ when $\mathcal{L} = \{l_0, l_1, l_2, ...\}$ is a finite set of labels. FREQT is an enhanced version of Apriori algorithm [22] that finds frequent itemsets in given transactions. FREQT receives a labeled ordered tree t and minimum support σ ($0 \leq \sigma \leq 1$) as input. $f(p,t)$ means a frequency of tree pattern p in t. More concretely, $f(p,t)$ is the number of times that p appears in t divided by the number of nodes in t. If $\sigma \leq f(p,t)$ is satisfied, FREQT outputs p as a frequent tree pattern.

FREQT finds $\mathcal{F}_{k+1}$ (a set of frequent tree patterns including k+1 nodes) by using $\mathcal{F}_k$ (a set of frequent tree patterns including k nodes). Its algorithm consists of four steps.

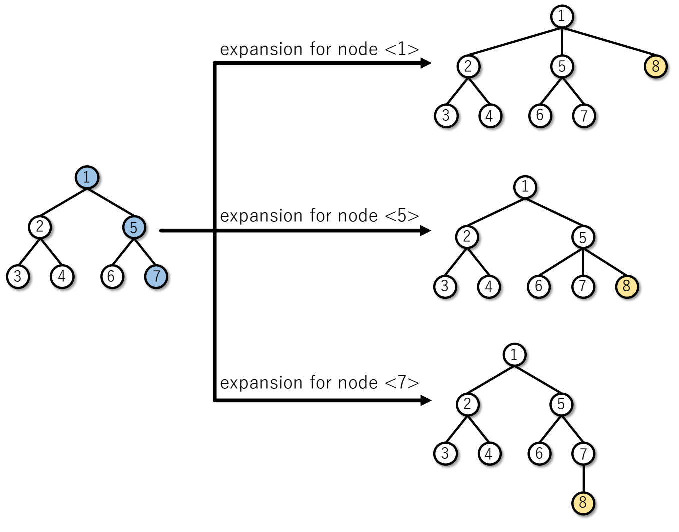


Fig. 2. Example of *rightmost expansion*. The blue nodes represent the rightmost branch, and the yellow nodes represent that they have been added by *rightmost expansion*

STEP-A. FREQT scans the nodes of a given labeled ordered tree t to find $\mathcal{F}_1$, which is a set of frequent nodes.

STEP-B. FREQT enumerates candidate tree patterns of size k+1 by using $\mathcal{F}_k$ and FREQT adds them to $\mathcal{C}_{k+1}$.

STEP-C. FREQT calculates $f(c,t)$, which is a frequency of candidate tree $c \in \mathcal{C}_{k+1}$ in t. If $\sigma \leq f(c,t)$ is satisfied, FREQT adds c to $\mathcal{F}_{k+1}$.

STEP-D. FREQT terminates if $\mathcal{F}_{k+1}$ is an empty set. If not, k is incremented by 1 and FREQT goes back to STEP-B.

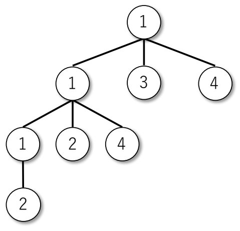
When enumerating candidate tree patterns of size k+1 from $\mathcal{F}_k$ in STEP-B, it is inefficient to add all labels $l \in \mathcal{L}$ for every node of every frequent tree pattern included in $\mathcal{F}_k$. Thus, FREQT performs *rightmost expansion* to efficiently generate candidate tree patterns.

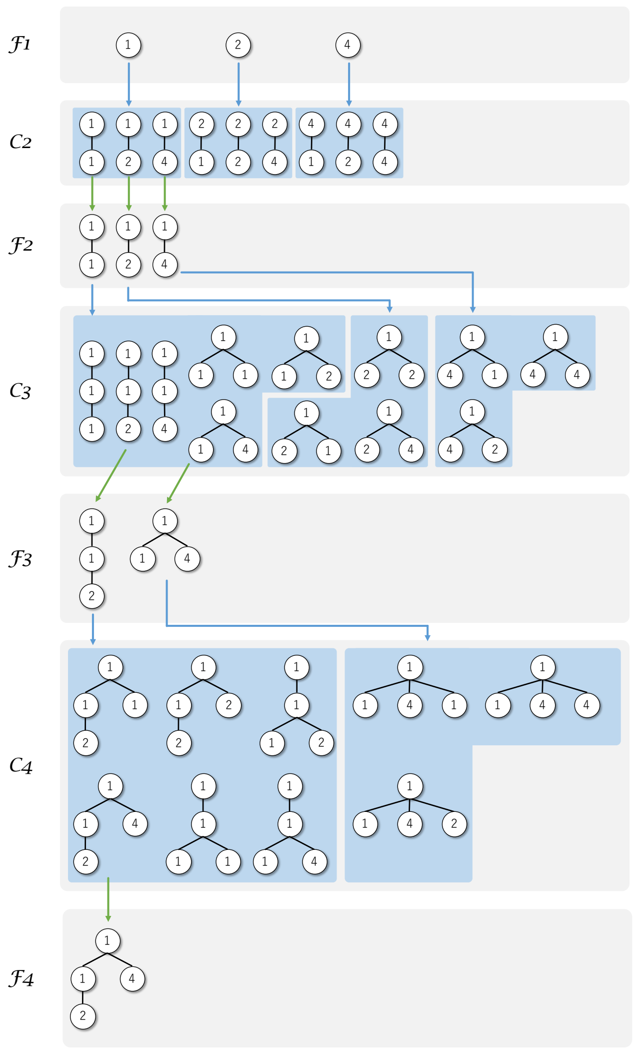
Before explaining *rightmost expansion*, we explain *rightmost leaf* and *rightmost branch*. A labeled ordered tree t is traversed with depth-first priority from its root node, and the last node to reach is *rightmost leaf*. The *rightmost branch* is the path from the root node to the *rightmost leaf*, and *rightmost expansion* means adding a child node to a node included in the *rightmost branch*. By performing *rightmost expansion*, the set of candidate trees $\mathcal{C}_{k+1}$ can be obtained without duplication or leakage.

Figure 2 shows an example of *rightmost expansion*. In this example, the *rightmost branch* includes nodes '1', '5', and '7', and for each node, node '8' is added. It is also inefficient to add a child node for every label $l \in L$ when doing *rightmost expansion*. Thus, FREQT takes the following heuristics.

Node-skip. FREQT adds only labels included in $\mathcal{F}_1$ in *rightmost expansion* because labels not included in frequent set $\mathcal{F}_1$ are never included in frequent set $\mathcal{F}_k$.

Edge-skip. Herein, an edge means a tree pattern whose size is 2, that is, a pair of a parent node and its child node. We assume that a label of the node selected for *rightmost expansion* is l. Only the labels of the child nodes of the edges whose parent node has label l and included in


(a) A labeled ordered tree



(b) Process of computing candidate/frequent tree patterns

Fig. 3. Example of how FREQT works

frequent tree patterns $\mathcal{F}_2$ are used in *rightmost expansion*.

Figure 3(b) shows an example of how candidate tree patterns $\mathcal{C}_k$ and frequent tree patterns $\mathcal{F}_k$ are computed for the labeled ordered tree shown in Figure 3(a). We can see that candidate tree patterns $\mathcal{C}_2$ are generated only for nodes included in $\mathcal{F}_1$ by performing *node-skip*. That is, node <3> is not used to compute $\mathcal{C}_2$.

IV. Proposed Technique

Figure 4 shows an overview of the proposed technique, TC2P. TC2P takes a Git repository as input and it generates tree-based change patterns contained in the repository. First, TC2P extracts commits in which at least a Java source file was changed in a given Git repository. Second, TC2P generates an edit script by utilizing GumTree for each of the changed Java files in the identified commits. Third, TC2P creates a *unified tree* by using the edit script and the before-change/after-change ASTs. A *unified tree* is a tree structure where a pair of before-change and after-change ASTs and an edit script between the two ASTs are combined into a single tree. We explain *unified tree* in Subsection IV-A in detail. After constructing *unified trees* from all the commits where at least a Java file is changed, TC2P performs a frequent tree pattern detection on the *unified trees* to obtain change patterns. Note that TC2P filters out frequent tree patterns that are not related to nodes included in the edit script.

In the remainder of this section, we explain two core analyses in TC2P, (1) constructing a *unified tree* tree from a changed file, and (2) detecting frequent tree patterns from *unified trees* in Subsections IV-A and IV-B, respectively. Then, we describe our implementation in Subsection IV-D.

A. *Constructing a unified tree*

The reason why TC2P constructs *unified trees* is to consider relative positions of nodes included in edit scripts in detecting change patterns. By examining an edit script generated by GumTree, TC2P identifies which nodes in the before-change and after-change ASTs have been manipulated. A *unified tree* construction includes the following processing.

- TC2P collects all nodes that were manipulated in a given edit script.
- TC2P deletes unnecessary nodes.

First of all, TC2P collects all nodes that were manipulated in a given edit script. The collecting processing is performed with the following algorithm.

- For *insert* operations, TC2P adds the inserted nodes into the before-change AST. Nodes are added to the same places as the after-change AST.
- For *delete* and *update* operations, TC2P does nothing because the deleted and updated nodes are already included in the before-change AST.
- For *move* operations, TC2P adds the moved subtrees into the before-change AST at the same place as the after-change AST. This means that, for each *move* operation, the subtree of the *move* target exists at two places: where it existed before the *move* operation and where it exists after the *move* operation.

Consequently, a *unified tree* means the before-change AST where target nodes of *insert* operations and moved subtrees of *move* operations have been added. A *unified tree* after the above processing includes many nodes that are not manipulated in the edit script. If we detect frequent tree patterns from such native *unified trees*, we will obtain many tree patterns that are not related to the edit script. Thus, we delete unnecessary nodes from naive *unified trees*. The nodes not satisfying any of the following conditions are deleted:

- included in the targets of the edit operations,
- included in the *move* subtrees,
- having two or more child nodes, and

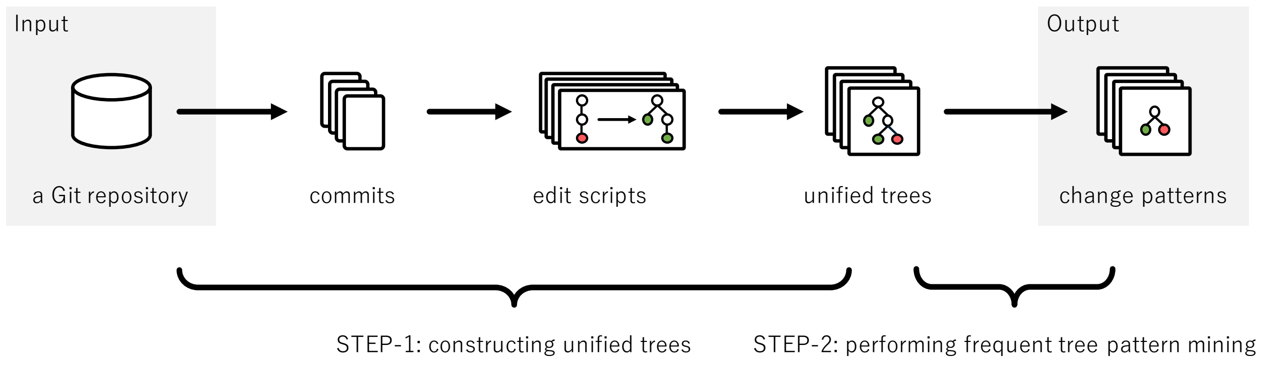

Input
a Git repository
commits
edit scripts
unified trees
Output
change patterns
STEP-1: constructing unified trees
STEP-2: performing frequent tree pattern mining

Fig. 4. Overview of TC2P

- having a parent node.

Figure 5(a) is a short edit script that includes all the four types of operations and Figure 5(b) shows how a *unified tree* is constructed based on the edit script. The inserted nodes '`15`' and '`16`' and the moved subtree '`6`' in the after-change AST are added to the before-change AST, which we call naive *unified tree*. Then, nodes '`0`', '`1`', '`2`', '`3`', '`10`', '`11`', and '`14`' are deleted because they do not satisfy any of the above conditions. By eliminating unnecessary nodes, the size of the *unified tree* is decreased to 13 from 20 in this example.

B. Detecting frequent tree patterns

After constructing *unified trees*, TC2P mines them to detect frequent tree patterns. In this processing, TC2P utilizes FREQT [15] with a little expansion. While original FREQT detects frequent tree patterns in a single tree, TC2P need to detect them in multiple trees. Consequently, we changed the algorithm of FREQT as follows.

- In STEP-A (, which is described in Subsection III-C), original FREQT scans a single tree to obtain $\mathcal{F}_1$, but in the expanded version, all the multiple trees are scanned to obtain $\mathcal{F}_1$.
- In original FREQT, the minimum support σ is set as $0 \leq \sigma \leq 1$, and tree patterns appearing at a rate greater than or equal to σ are regarded as frequent tree patterns. However, in detecting change patterns, it is more natural to base the number of occurrences of change patterns rather than the rate of their occurrences. In fact, prior studies used the number of occurrences, not the rate of occurrences [14], [23]. Therefore, in TC2P, the minimum support σ of change pattern is set as $1 \leq \sigma$, and tree patterns with more than σ times of occurrences are regarded as frequent tree patterns.

To detect frequent tree patterns, it is necessary to define node equivalence in *unified trees*. In TC2P, two nodes are defined as equivalent if they satisfy all the following conditions.

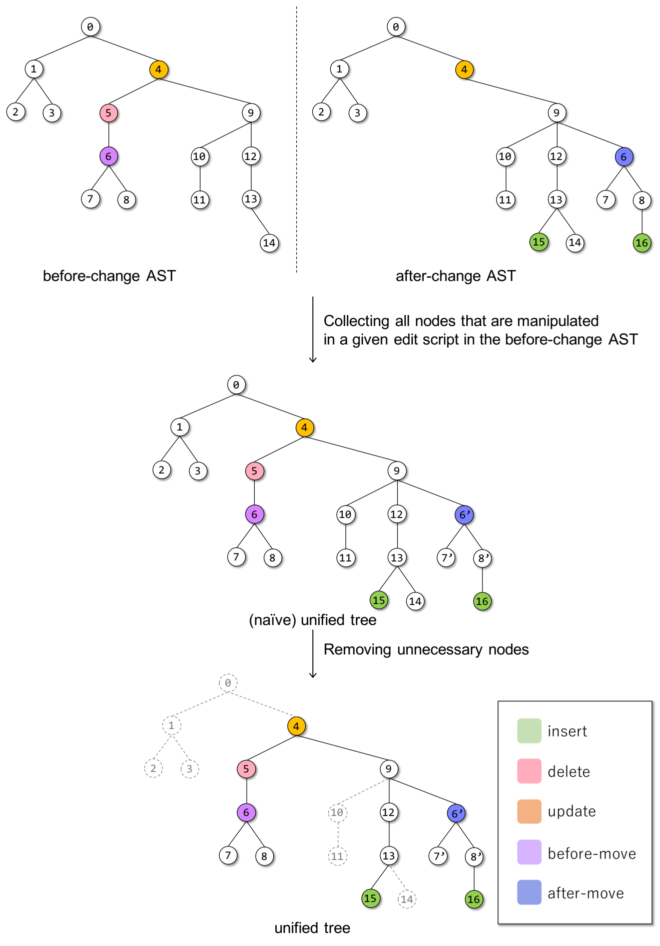
- Types of edit actions for the two nodes are the same.
- Labels of the two nodes are the same.
- If the two nodes include values, the values are the same.
- If the edit actions for the nodes are *update*, the updated values are the same.

TC2P includes a new mechanism to reduce computational costs in detecting frequent tree patterns. TC2P does not use

```
update(4, *)
move(6, 9, 3)
insert(15, 13, 1, *, *)
insert(16, 8, 1, *, *)
delete(5)
```

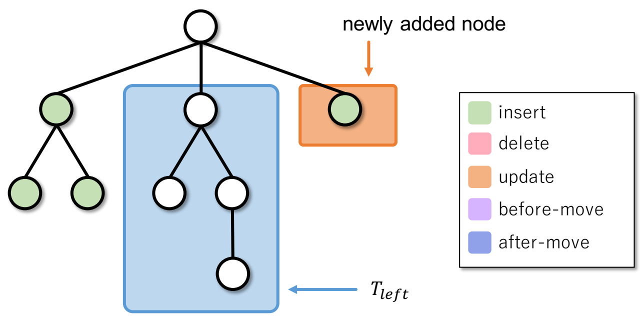
Asterisks indicate labels or values that do not need to be considered here

(a) A given edit script

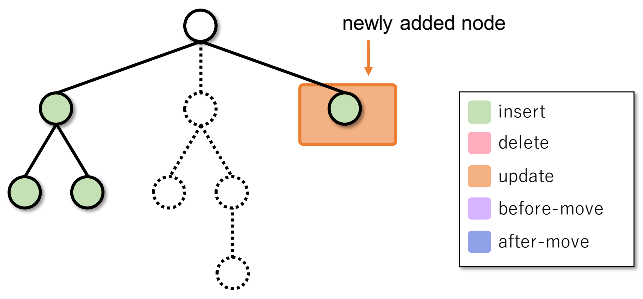

before-change AST
after-change AST
Collecting all nodes that are manipulated in a given edit script in the before-change AST
(naïve) unified tree
Removing unnecessary nodes
unified tree
insert
delete
update
before-move
after-move

(b) Construction Procedure

Fig. 5. An example of constructing a unified tree



(a) Useless Tree Pattern



(b) Useful Tree Pattern

Fig. 6. Examples of useful and useless tree patterns

a frequent tree pattern of size k as a base of candidate tree patterns of size k+1 if it is the base of only *useless* change patterns. Figure 6(a) shows that a new node has been added to a frequent tree pattern included in $\mathcal{F}_8$ by performing *rightmost expansion*. Figure 6(b) shows a frequent tree pattern included in $\mathcal{F}_4$ and a new node has been added to it by performing *rightmost expansion*. The difference between the tree pattern in $\mathcal{F}_8$, and the one in $\mathcal{F}_4$ is whether subtree T_{left} is included or not. For this tree pattern of size nine in Figure 6(a), another node will be added to the root node or the new node because TC2P performs *rightmost expansion*. This means that subtree T_{left} does not include manipulated nodes at this moment, and it will never include such manipulated nodes even in larger tree patterns. All change patterns found by using the change pattern of size nine in Figure 6(a) are always based on the change pattern of size four in Figure 6(b). Consequently, TC2P does not add the change pattern of size nine to $\mathcal{C}_9$. This filtering can save unnecessary computational costs to detect frequent change patterns. More concretely, if a change pattern of size k includes subtree T_{left} that satisfies both following conditions, the change pattern is not added to $\mathcal{C}_k$.

- T_{left} is not included in any of the subtrees for *move* operations.
- T_{left} does not include any target node of *insert*, *delete*, and *update* operations.

C. Method-level AST Comparison

An AST is usually constructed for each file. Since a Java source file can include multiple methods, a *unified tree* can include subtrees of multiple methods. Therefore, when two or more methods in a file are accidentally changed in the same commit, A single change pattern will be detected from the *unified tree*. For this reason, our technique constructs *unified trees* from method-level ASTs instead of file-level ASTs. By using method-level ASTs, our technique avoids generating a single change pattern from changes made to multiple methods in the same commit by chance.

D. Implementation

We have implemented TC2P as a software tool. At this moment, TC2P is implemented for Java language, but it is not difficult to expand TC2P for other programming languages that are supported by GumTree. Hereafter, we call our implementation TC2P.

TC2P utilizes SQL database as a cache for *unified trees*. When a commit is given to TC2P, it checks whether *unified trees* of the given commits are in the database. Only if *unified trees* are not registered for the given commit, TC2P constructs *unified trees* from the commit and registers them to the database. This caching mechanism works well if we run TC2P on the same repositories again and again. Besides, TC2P performs the processing of detecting frequent tree patterns $\mathcal{F}_k$ with the customized FREQT (described in Subsection IV-B) in parallel.

In Section IV-C, we explained that our technique constructs a *unified tree* for each method. To construct a *unified tree* for each method, we need to track each method instead of each file. To retrieve method histories, we use FinerGit [24], which is a tool that converts a Git repository of Java project to a finer-grained one. In a converted repository, each Java method is preserved as a single file. Thus, we can easily retrieve method histories only by using Git commands such as git-log.

V. Evaluation

We evaluate TC2P in terms of the following two points:

- quality of change patterns detected by TC2P, and
- execution time of TC2P.

A. Dataset

We take advantage of the dataset published by Borges et al. [25] in this evaluation. The dataset includes 202 repositories of Java projects[3]. We use method histories instead of file histories because TC2P is designed to utilize method histories (described in Subsection IV-C). To retrieve method histories, we apply FinerGit to the 202 repositories to generate their finer-grained repositories. The 202 projects vary in size, but in total they include 1.3M commits, 32.9M lines of code, and 1.8M methods. We use the latest 1,000 commits in each of the 202 finer-grained repositories to detect change patterns.

B. Re-implementation of Negara's technique

We evaluate the qualify of change patterns detected by TC2P by comparing them with Negara's technique [14]. However, the tool and the dataset that they used in literature [14] are not open to the public. Thus, we implemented another tool NGR based on the literature. NGR mines a set of edit scripts and derives change patterns by using itemset mining [26].

The equivalence of edit operations in NGR is different from TC2P because NGR does not consider the tree structure of change patterns. In NGR, given two edit operations are defined

[3]https://goo.gl/73Sbvz

as equivalent if the two edit operations satisfy the following conditions:

- types of the edit actions in the two edit operations are the same (e.g., insertion, deletion), and
- labels of the target nodes in the edit operations are the same (e.g., if-statement, return-statement).

In this evaluation, we detect change patterns with TC2P and NGR from the fine-grained repositories. By applying both tools to method histories, we can compare the differences between tree-based mining and text-based one.

C. Configurations

We conducted the following two experiments.

EXP-1. We compare change patterns detected by TC2P with ones detected by NGR for each of the target repositories.

EXP-2. We measure the execution time of TC2P with different support values. A support value means a minimum occurrence number of change patterns that TC2P detects.

In the experiments, the time limit for all executions of TC2P and NGR is set to two hours. We run all the experiments on a single workstation equipped with 64GB memory and two 12-core CPUs. TC2P and NGR are implemented as multi-threaded programs to make good use of the resource of the workstation.

In EXP-1, we detect change patterns by using TC2P and NGR for each of the target repositories. For TC2P, we use ten as the minimum support value. For NGR, we use the values in Table I for Absolute Frequency Threshold (in short, AFT) and Dynamic Threshold (in short, DT). A candidate change pattern whose number of occurrences does not exceed the AFT is not treated as a change pattern. If the size of the candidate pattern multiplied by the number of occurrences does not exceed DT, it is not treated as a change pattern. NGR enumerates candidate change patterns by adding edit operations one by one like Apriori algorithm [22] with dynamically switching the thresholds between AFT and DT depending on the number of occurrences of the added edit operations. This algorithm is based on the descriptions in literature [14].

In EXP-2, for each repository, we run TC2P with different minimum support values and measure the execution time. We use 10, 25, 50, and 100 as the thresholds.

D. Results of EXP-1

To check whether TC2P can detect change patterns more adequately than NGR, we compare change patterns detected by the two tools. However, it is impossible to directly compare change patterns detected by the two tools for the following reason. Each change pattern detected by TC2P is a pattern of tree structure while each change pattern detected by NGR is a set of edit operations. Consequently, in this experiment, we convert each of the change patterns detected by TC2P to a set

TABLE I
PARAMETERS FOR BASE

Frequency $\mathcal{F}$	AFT	DT
$250 \leq \mathcal{F}$	10	250
$50 \leq \mathcal{F} < 250$	10	50
$10 \leq \mathcal{F} < 50$	10	10

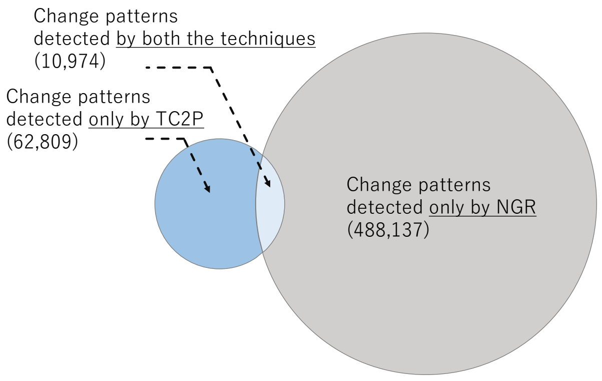


Fig. 7. The number of change patterns detected by TC2P and/or NGR

of edit operations. Then, the converted sets are compared with change patterns detected by NGR. If a set of edit operations converted from a tree pattern is exactly the same as a set of edit operations in a change pattern detected by NGR, we regard the TC2P's change pattern (a tree structure) is the same as the NGR's change pattern (a set of edit operations).

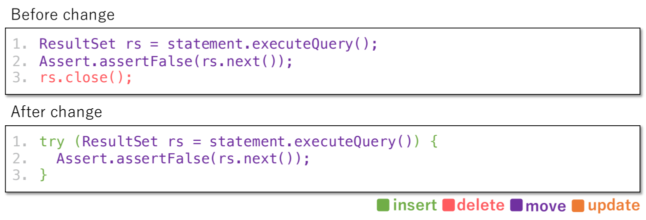
By converting the tree structure of each change pattern detected by TC2P to a set of edit operations, the information of relative positions of nodes manipulated in the change patterns is lost. However, we can compare the detection results of the two techniques. Figure 7 shows the comparison results with a Venn diagram. The numbers of change patterns detected only by TC2P or only by NGR are 62,809 and 488,137, respectively. The two techniques share only 10,974 change patterns.

Firstly, we focus on the change patterns detected only by TC2P. We found that 62% of those change patterns included at least a *move* action. NGR does not consider that a code fragment is moved to another place. In NGR's change patterns, a moved code fragment is represented by a sequence of *delete* operations and another sequence of *insert* operations. On the other hand, TC2P has the capability of identifying code moves. Representing moved code fragments with *move* actions is definitely more adequate than representing them with *delete* and *insert* actions.

For the remaining 38% change patterns, the reason why only TC2P detected them was due to different kinds of thresholds between the two techniques. TC2P adopts a simple static threshold, and we use ten as it. On the other hand, NGR takes dynamic thresholds, and we use the values in Table I based on literature [14]. Although it is possible to minimize the detection of such change patterns by tuning the parameters, we did not do so because it is not the main purpose of this experiment.

Next, we focus on the change patterns detected only by NGR. Those change patterns account for 97% of the change patterns detected by NGR. We randomly selected 30 change patterns from the top 1,000 change patterns with many edit operations that were detected only by NGR and visually browsed their change instances.

There are 29 change patterns whose change instances include different tree structures from other change instances in the same change pattern. TC2P considers relative positions


Before change

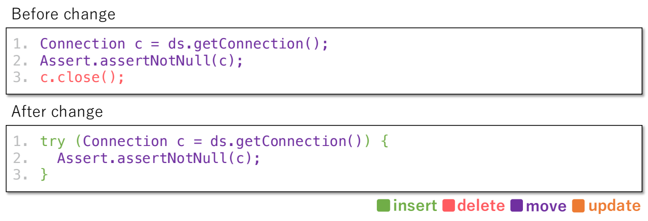
```
ResultSet rs = statement.executeQuery();
Assert.assertFalse(rs.next());
rs.close();
```

After change

```
try (ResultSet rs = statement.executeQuery()) {
  Assert.assertFalse(rs.next());
}
```

🟩insert 🟥delete 🟪move 🟧update

(a) An actual change for ResultSet


Before change

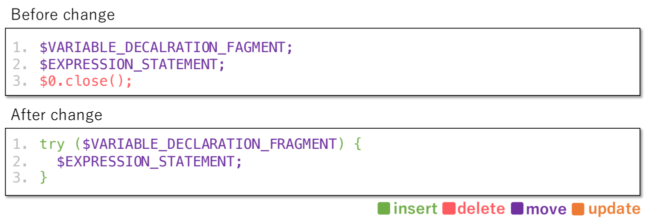
```
Connection c = ds.getConnection();
Assert.assertNotNull(c);
c.close();
```

After change

```
try (Connection c = ds.getConnection()) {
  Assert.assertNotNull(c);
}
```

🟩insert 🟥delete 🟪move 🟧update

(b) An actuual change for Connection


Before change

```
$VARIABLE_DECALRATION_FAGMENT;
$EXPRESSION_STATEMENT;
$0.close();
```

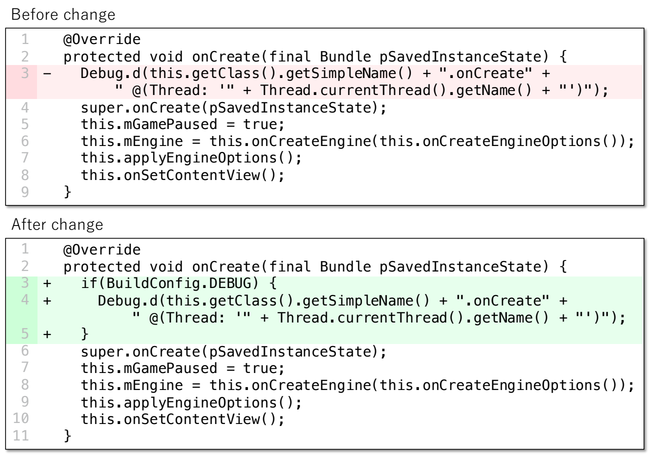
After change

```
try ($VARIABLE_DECLARATION_FRAGMENT) {
  $EXPRESSION_STATEMENT;
}
```

🟩insert 🟥delete 🟪move 🟧update

(c) The detected pattern

Fig. 8. Two changes and their change pattern related to try block

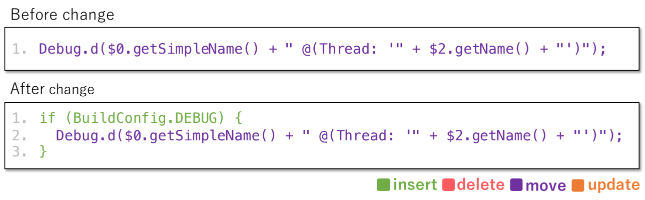

Before change

```
  @Override
  protected void onCreate(final Bundle pSavedInstanceState) {
-   Debug.d(this.getClass().getSimpleName() + ".onCreate" +
          " @(Thread: '" + Thread.currentThread().getName() + "')");
    super.onCreate(pSavedInstanceState);
    this.mGamePaused = true;
    this.mEngine = this.onCreateEngine(this.onCreateEngineOptions());
    this.applyEngineOptions();
    this.onSetContentView();
  }
```

After change

```
  @Override
  protected void onCreate(final Bundle pSavedInstanceState) {
+   if(BuildConfig.DEBUG) {
+     Debug.d(this.getClass().getSimpleName() + ".onCreate" +
            " @(Thread: '" + Thread.currentThread().getName() + "')");
+   }
    super.onCreate(pSavedInstanceState);
    this.mGamePaused = true;
    this.mEngine = this.onCreateEngine(this.onCreateEngineOptions());
    this.applyEngineOptions();
    this.onSetContentView();
  }
```

(a) An actual change


Before change

```
Debug.d($0.getSimpleName() + " @(Thread: '" + $2.getName() + "')");
```

After change

```
if (BuildConfig.DEBUG) {
  Debug.d($0.getSimpleName() + " @(Thread: '" + $2.getName() + "')");
}
```

🟩insert 🟥delete 🟪move 🟧update

(b) The detected pattern

Fig. 9. A change and its change pattern related to logging

of nodes manipulated in edit operations while NGR does not, which is the reason why those change patterns were not detected by TC2P, as shown in Figure 1.

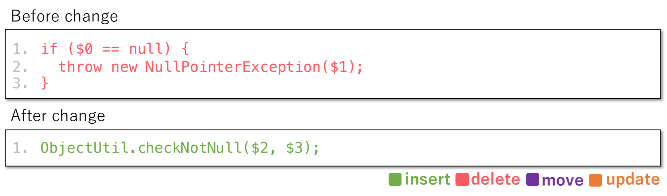
The remaining change pattern included code changes whose tree patterns are the same. The corresponding change pattern in TC2P included a *move* action, which is the reason why TC2P does not the exactly same change pattern as NGR. The investigation results show that the change patterns detected only by NGR are also detected by TC2P in more adequate forms or not detected by TC2P due to the structural inconsistencies. Consequently, we conclude that there is no change pattern to be detected as they are in the 30 change patterns detected only by NGR.

Herein, we show some change patterns detected by TC2P. Figure 9 shows a change for debugging output and its change pattern. Figure 9(a) shows an instance of the changes forming the change pattern. We can see that the statement of "`Debug.d(...`" gets surrounded by the if-statement in the change. Figure 9(b) shows the derived change pattern. TC2P generates change patterns as tree structures (In the figure, the detected pattern is shown with the source code). In the figure, the variables whose nodes are not included in the change pattern are represented by "$ + number". In the pattern, we can see that the statement of "`Debug.d(...`" has been moved into the newly added if-statement. As already mentioned, *move* operation is not supported by NGR. If a code move exists in a given change, NGR output it as a sequence of *delete* operations and another sequence of *insert* operations. Moreover, NGR does not consider relative positions of manipulated nodes in edit scripts in detecting change patterns. For the above two reasons, it is impossible to know where a node has moved from to where with NGR's change pattern. In TC2P, on the other hand, the change pattern is detected by using the information of the AST structure. By using TC2P's change pattern, we can see where a node has moved from to where, as shown in Figure 9(b). As shown in this example, we can understand that *move* action is more compatible with tree-based change patterns.

Next, we show a change pattern detected only by TC2P. Figure 8(c) shows a change pattern where a try-statement is added. `$VARIABLE_DECLARARION_FRAGMENT` represents a variable-declaration, and `$EXPRESSION_STATEMENT` represents an expression-statement. In the change pattern, the following manipulations are conducted.

- The variable-declaration is moved into the clause of the try-statement.
- The expression-statement is moved into the block of the try-statement.
- The statement of "`$0.close()`" is deleted.

Herein, we focus on the fact that *move* actions have been applied to the abstracted AST nodes, not to actual program elements, in the change pattern. Because TC2P applies frequent tree pattern mining to ASTs, it can abstractly maneuver them as change patterns regardless of the concrete processing such as what kinds of variables are declared. Figures 8(a) and 8(b) show two actual changes included in the change pattern. The types of the declared variables in the moved variable-declarations and the invoked methods in the moved expression-statements are different between the two changes. Even though those two changes contain such differences, TC2P was able


Before change

```
if ($0 == null) {
  throw new NullPointerException($1);
}
```

After change

```
ObjectUtil.checkNotNull($2, $3);
```

Fig. 10. A change pattern related to null checking

to detect the change pattern common to them. This is only possible by using the tree structure to detect the change pattern.

Figure 10 shows a change pattern where EXTRACT METHOD refactoring is applied to a null-checking code fragment. In the change pattern, the invoked constructor name (`NullPointerException`) and the method name (`checkNotNull`) is not abstracted unlike Figure 8(c). Concrete names in the change pattern mean that all the changes in the change pattern have the common names. Such a distinction between abstracted and concrete names is impossible in NGR, because NGR does not consider the parent-child relationship between nodes and cannot determine whether the manipulated nodes have a common parent or not.

With the experimental results, we conclude that TC2P can detect change patterns that are not detected by NGR. Furthermore, TC2P avoids detecting change patterns whose change instances have different tree structures by considering relative positions of nodes in each change instance. Consequently, we can conclude that quality of change patterns detected by TC2P is higher than ones detected by NGR.

E. Results of EXP-2

The results of EXP-2 are shown in Table II. The average execution time was calculated for projects where TC2P finishes detecting change patterns within two hours. The number of projects where TC2P execution exceeded the time limit is shown in the second column. TC2P has a longer time execution as the number of emerging change patterns increases. Thus, the larger the value of the minimum support is, the shorter the execution time because the number of subtrees treated as change patterns is reduced. For the minimum support of ten and 25, the average of 25 is shorter than the one of ten. But, in the case of ten, TC2P exceeded the time limit for 17 projects while only six projects in the case of 25.

Consequently, we conclude that TC2P was able to detect change patterns in an average of less than one minute for most target projects. However, it took more than two hours for some large projects.

TABLE II
EXECUTION TIME FOR EACH MINIMUM SUPPORT VALUE

minimum support	execution time (sec.)	# of time limit
10	30.3	17
25	38.8	6
50	11.9	2
100	4.8	0

F. Threats to Validity

Herein, we list some threats to validity in the experiment.

- Although all the experiments were conducted on Java projects, GumTree supports other programming languages than Java, such as Python and JavaScript. We may obtain different results for other programming language.
- We implemented NGR based on the information in literature [14], [27] as Negara's technique, and then we compared TC2P and NGR. If we misunderstood the specification of Negara's technique or induced non-trivial bugs to NGR, the comparison results do not make sense.

VI. RELATED WORK

A. Pattern Mining Algorithms

A variety of techniques have been proposed for mining patterns. Apriori [22] and Backtracking [28], [29] are algorithms to find frequent subsets from a set of multiple items. Apriori is a breadth-first search algorithm, and Backtracking is a depth-first one. PrefixSpan [30] has been proposed as an algorithm for finding frequent subsequences from multiple sequences. PrefixSpan is used in a study on mining patterns in the source code [31], [32], [33].

In addition to FREQT [15], TreeMiner [34] is another algorithm for mining frequent tree patterns. TreeMiner, like FREQT, performs *rightmost expansion* to find frequent subtrees. However, unlike FREQT, TreeMiner mines subtrees based on whether given two nodes are a pair of an ancestor and its descendant or not, rather than whether given two nodes are pair of a parent and its direct child or not, when searching for frequent tree patterns. That is, TreeMiner finds frequent tree patterns containing different parts. In this research, to avoid such frequent tree patterns in *unified trees*, we utilize FREQT.

B. Identifying AST Differences

ChangeDistiller [35] is an algorithm for calculating the differences in ASTs in addition to GumTree. GumTree and ChangeDistiller were compared in literature [16], and the comparison results are as follows.

- GumTree was able to identify code moves more accurately than ChangeDistiller.
- ChangeDistiller tended to produce longer edit scripts than GumTree.

For the above reasons, we selected to use GumTree in this research.

C. Detecting Change Patterns

The work of Negara et al. [14] is already described in Section II. Herein, we describe some other techniques to detect change patterns. CPatMiner presents code changes with program dependence graphs and mines graph patterns to detect change patterns [23]. Figure 11 shows the change described in literature [23] as an example of how CPatMiner successfully detected change patterns. For the change, CPatMiner was able to find a pattern that "`statement.close();`" was replaced with "`closeQuietly(statement);`" by utilizing the data dependence for "`statement`". On the other hand, TC2P can identify a change pattern that including the following manipulations because it considers the tree structure.

- A try-block and a finally-block were inserted.
- The statements of lines 2∼9 were moved into the try-block.
- "`statement.close();`" was deleted and "`closeQuietly(statement);`" was inserted.

By using this example, we can conclude that TC2P is superior to CPatMiner in the following points.

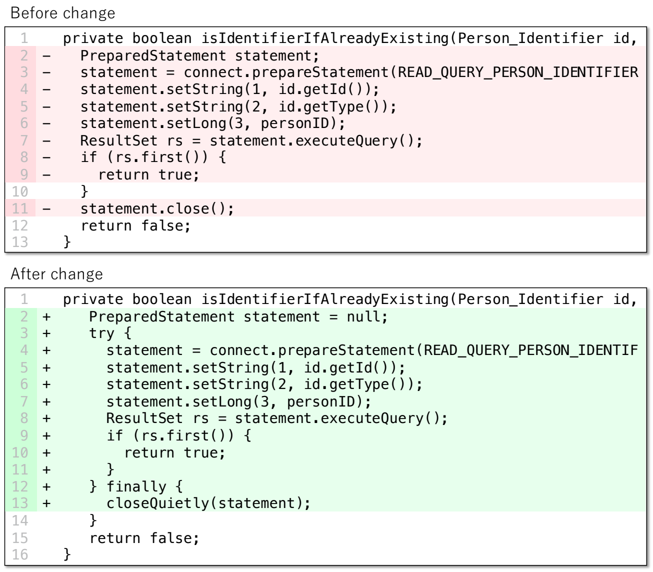
- TC2P can capture the change of any kind of inner block insertions and deletions by using parent node information. On the other hand, CPatMiner can capture the change of only conditional block insertions and deletions by using control dependences. In Java programs, there are many change patterns that are related to try-blocks as shown in Figures 8 and 11. Only TC2P can detect such change patterns as try-block related change patterns. CPatMiner cannot detect them as try-block related ones because try-blocks and their inner statement have no control dependences.
- TC2P can capture code moves by using the parent node information of the manipulated nodes while CPatMiner technique cannot.

On the other hand, CPatMiner is superior to TC2P in the following point.

- CPatMiner can capture statement-level replacements by using data and control dependencies while such changes are captured as deletions and insertions in TC2P.

There are other techniques that detect change patterns. In Nguyen et al.'s work [4], only subtrees with the same height are used as candidates for change patterns. In Martinez et al.'s work [36], change patterns whose only a single subtree is modified are detected. In other words, if two different subtrees in ASTs are manipulated in a change, a change pattern including the two manipulations is not detected. That is, both the techniques have certain constraints in detecting change patterns while TC2P does not have such constraints.

There are some techniques to detect change patterns in method invocations [6], [12], [37]. Those techniques are specialized to detect method invocation patterns, and they cannot detect general change patterns. Coming [13] is a technique to mine instances of specific change patterns from Git repositories. This technique mines changes by using predefined change patterns while TC2P detects unknown ones. LASE is a technique that generates common change pattern from given multiple change instances [38]. This technique is not suited for detecting unknown changes from commit histories.



Before change

```
1     private boolean isIdentifierIfAlreadyExisting(Person_Identifier id,
2  -    PreparedStatement statement;
3  -    statement = connect.prepareStatement(READ_QUERY_PERSON_IDENTIFIER
4  -    statement.setString(1, id.getId());
5  -    statement.setString(2, id.getType());
6  -    statement.setLong(3, personID);
7  -    ResultSet rs = statement.executeQuery();
8  -    if (rs.first()) {
9  -      return true;
10      }
11 -    statement.close();
12      return false;
13    }
```

After change

```
1     private boolean isIdentifierIfAlreadyExisting(Person_Identifier id,
2  +    PreparedStatement statement = null;
3  +    try {
4  +      statement = connect.prepareStatement(READ_QUERY_PERSON_IDENTIF
5  +      statement.setString(1, id.getId());
6  +      statement.setString(2, id.getType());
7  +      statement.setLong(3, personID);
8  +      ResultSet rs = statement.executeQuery();
9  +      if (rs.first()) {
10 +        return true;
11 +      }
12 +    } finally {
13 +      closeQuietly(statement);
14      }
15      return false;
16    }
```

Fig. 11. "Closing resources" pattern in project anHALytics/anhalytics-core

VII. Conclusion

We proposed a technique for detecting change patterns in this paper. Our technique constructs a *unified tree* from an edit script generated by GumTree, and then it mines frequent tree patterns in the *unified trees*. We treat detected frequent tree patterns as tree-based change patterns in this research. We have implemented a software tool, TC2P, based on the proposed technique, and we evaluated it on 202 open source projects. In the evaluation, we compared TC2P with two prior techniques, Negara's technique [14] and Nguyen's technique [23]. The former detects change patterns by finding frequent edit operations in given edit scripts, and the latter detects change patterns by finding isomorphic subgraphs in given program dependence graphs. Through the evaluation, we confirmed that TC2P was able to detect change patterns more adequately.

In the future, we are going to use TC2P with the following situations.

- We are going to use TC2P to find inconsistencies in the source code. Several studies have already conducted inconsistency detections [8], [39], [40], [41], but they used text-based or graph-based patterns. We confirmed that TC2P was able to detect change patterns more adequately than text-based and graph-based techniques in this paper. Thus, we believe new inconsistencies in the source code can be revealed with TC2P.
- Edit scripts generated by GumTree consist of basic edit operations such as inserting, deleting, moving, and updating AST nodes. If we could name TC2P's change patterns by using tree structure information, we may able to define new higher-level edit operations. Higo et al. defined a special sequence of *insert* actions as a *copying-and-pasting* operation by using GumTree [42]. Naming higher-level edit operations will make it easier for developers to understand given changes.

Acknowledgments

This work was partially supported by MEXT/JSPS KAKENHI JP20H04166.

References

[1] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 306–317.

[2] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 837–847.

[3] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, 2009, pp. 81–92.

[4] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 180–190.

[5] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009, pp. 213–222.

[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 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 511–522.

[7] R. Robbes and M. Lanza, "How program history can improve code completion," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 317–326.

[8] Y. Higo, S. Hayashi, H. Hata, and M. Nagappan, "Ammonia: an approach for deriving project-specific bug patterns," *Empirical Software Engineering*, 2020.

[9] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 613–624.

[10] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 802–811.

[11] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *Proceedings of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 213–224.

[12] G. Uddin, B. Dagenais, and M. P. Robillard, "Analyzing temporal api usage patterns," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 456–459.

[13] M. Martinez and M. Monperrus, "Coming: A tool for mining change pattern instances from git commits," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, 2019, pp. 79–82.

[14] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 803–813.

[15] T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, and S. Arikawa, "Efficient substructure discovery from large semi-structured data," *IEICE TRANSACTIONS on Information and Systems*, vol. 87, no. 12, pp. 2754–2763, 2004.

[16] 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*, 2014, pp. 313–324.

[17] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 493–504.

[18] C. Macho, S. Mcintosh, and M. Pinzger, "Extracting build changes with builddiff," in *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017, pp. 368–378.

[19] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 25–36.

[20] 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 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 740–751.

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

[22] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pp. 487–499.

[23] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 819–830.

[24] Y. Higo, S. Hayashi, and M. Nagappan, "On tracking java methods with git mechanisms," *Journal of Systems and Software*, 2020.

[25] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution*, 2016, pp. 334–344.

[26] M. J. Zaki and C.-J. Hsiao, "Charm: An efficient algorithm for closed itemset mining," in *Proceedings of the 2002 SIAM International Conference on Data Mining*, 2002, pp. 457–473.

[27] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining continuous code changes to detect frequent program transformations," Tech. Rep., 2013. [Online]. Available: http://hdl.handle.net/2142/43889

[28] R. J. Bayardo, "Efficiently mining long patterns from databases," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 1998, pp. 85–93.

[29] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for fast discovery of association rules," in *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, 1997, pp. 283–286.

[30] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth," in *Proceedings of the 17th International Conference on Data Engineering*, 2001, pp. 215–224.

[31] H. Date, T. Ishio, and K. Inoue, "Investigation of coding patterns over version history," in *Proceedings of the 2012 Fourth International Workshop on Empirical Software Engineering in Practice*, 2012, pp. 40–45.

[32] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 254–265.

[33] Y. Zhang, Y. Liu, L. Zhang, and Y. Shi, "A data mining based method: Detecting software defects in source code," in *Proceedings of the 2nd International Conference on Software Engineering and Data Mining*, 2002, pp. 607–612.

[34] M. J. Zaki, "Treeminer: An efficient algorithm for mining embedded ordered frequent trees," in *Advanced Methods for Knowledge Discovery from Complex Data*. Springer, 2005, pp. 123–151.

[35] 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.

[36] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with ast analysis," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, 2013, pp. 388–391.

[37] J. Andersen and J. L. Lawall, "Generic patch inference," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 337–346.

[38] 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. IEEE Press, 2013, pp. 502–511.

[39] S. Kim, K. Pan, and E. E. J. Whitehead, "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 35–45.

[40] G. Liang, Q. Wang, T. Xie, and H. Mei, "Inferring project-specific bug patterns for detecting sibling bugs," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 565–575.

[41] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 315–324.

[42] 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*, 2017, pp. 532–542.