Search-Based Refactoring Detection from Source Code Revisions

Shinpei HAYASHI†, Yasuyuki TSUDA††, Nonmembers, and Motoshi SAEKI†, Member

SUMMARY This paper proposes a technique for detecting the occurrences of refactoring from source code revisions. In a real software development process, a refactoring operation may sometimes be performed together with other modifications at the same revision. This means that detecting refactorings from the differences between two versions stored in a software version archive is not usually an easy process. In order to detect these impure refactorings, we model the detection within a graph search. Our technique considers a version of a program as a state and a refactoring as a transition between two states. It then searches for the path that approaches from the initial to the final state. To improve the efficiency of the search, we use the source code differences between the current and the final state for choosing the candidates of refactoring to be applied next and estimating the heuristic distance to the final state. Through case studies, we show that our approach is feasible to detect combinations of refactorings.

key words: refactoring, version archives, source code differences, informed search, heuristics

1. Introduction

This paper proposes a technique for detecting the occurrences of refactoring [1, 2] from two versions of source code stored in a software version archive. Refactorings are changes acted internally to improve the maintainability of source code. Refactorings are named, typical, and well-known modifications. So developers can easily understand how refactorings modify the program from their names and contexts. However, the information of performed refactorings is often not recorded and missed for various reasons. Detecting refactorings from version archives such as CVS [3] or Subversion [4] is essential for achieving useful supports such as capturing and understanding the changes acted internally to improve the maintainability of code stored in a software version archive. Refactorings are not satisfied by other modifications made.

To detect these impure refactorings [8], made in-between the program before and that after modifications (in short pre/post-modified programs), we propose a novel technique that utilizes a graph search, which is commonly applied within the field of Artificial Intelligence. In this technique, a version of the program is assumed as a state and a refactoring as a transition operator. The goal is to find the path that approaches from the initial to the final state. Furthermore, in order to improve the efficiency of the search, we focus on the differences in the source code between the current and the final state for choosing the candidates of refactoring to be applied next and estimating the heuristic distance to the final state. The proposed method was applied for two Java programs and has shown its effectiveness from the experiments that were conducted.

For the starting point to efficiently detect refactorings from software version archives, we focus on the problems for detecting impure refactorings. The key techniques used in our approach are as follows, for efficiently detecting impure refactorings:

Considering refactoring detection as a graph search. To detect impure refactorings, we have to consider intermediate hidden states between two versions of a program. In order to re-generate these states, we actually perform refactoring operations to the old version of the program and search for an appropriate path between two states by using a graph search algorithm.

A heuristic function based on program differences. To achieve an efficient search, it is necessary to reduce the search space and to generate candidates of possible refactorings. We use structural differences between two versions of a program for evaluating a state and deducing next states in the search.

The rest of this paper is organized as follows. In the next section, we describe a motivating example for detecting multiple occurrences of refactoring from a version archive. Section 3 outlines our methodology. Section 4 describes the automated tool that we have implemented. In Sect. 5, through case studies of refactoring detection, we show the feasibility of our approach. Section 6 discusses the most relevant work, and finally Sect. 7 concludes with our contributions and discusses an outline of our future plans.

2. Detecting Impure Refactorings

Refactoring is sometimes performed not as an individual operation but as a part of compound modification. For exam-
example, some refactorings become effective after another refactoring has already been performed. Furthermore, a prior-refactoring is often effective to thereafter correct the external behavior of the program. In general terms, a refactoring and other refactorings, and furthermore refactorings and non-refactoring modifications get entangled and appear in a program development history.

Under a configuration management using traditional version management systems such as CVS or Subversion, refactoring histories are prone to be erased because these systems mainly focus on just extracting prior-state of the revision sources. In these systems, the operation history, which contains what kind of refactorings were invoked, is not archived but is simply gathered by computing differences between two versions of source code stored in the version archives. In a real software development process, we sometimes fail to achieve fine-grained commits for wide varieties of reasons from, e.g., working costs or project/individual’s policies. As the result, a lot of refactoring operations are not archived properly, so we are forced to estimate the actual history based on consolidated differences output from traditional version management systems.

We state a motivating example of impure refactoring from Fowler’s refactoring book [1]. Figure 1 shows a case of refactoring activities that performs two refactoring operations: Extract Method and Move Method. First, a developer extracts a method amountFor from the body of a method statement in a class Customer by Extract Method refactoring (state #1 → #2). Actually, in the first stage, the following three modifications are adopted to the initial program:

- some statements are deleted in a method statement,
- amountFor is added to Customer, and
- a method invocation of amountFor is added to statement.

After this, in the second stage, Move Method is performed for amountFor and it is then moved to a class Rental (state #2 → #3), where the following three modifications are conducted:

- amountFor in Customer is removed,
- amountFor is added to Rental, and

```
public String statement() {
    ...
    while (rentals.hasMoreElement()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount = ...
        } ...
    }
}
```

```
public String statement() {
    ...
    while (rentals.hasMoreElement()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        thisAmount = amountFor(each);
    }
}
```

```
public double amountFor(Rental each) {
    double thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount = ...
    }
    return thisAmount;
}
```
the method invocation of amountFor in statement is updated.

These modifications are performed at once, so that if state #1 and #3 are stored without the state #2 in a version archive, the differences obtained from the version management system lacks state #2 as follows:

- Customer#statement is removed,
- a method invocation to Rental#amountFor is added to Customer#statement, and
- Rental#amountFor is added.

From the difference without considering state #2, one could only obtain the limited information.

These situations show an occurrence of a compound refactoring where other refactorings exists in the same revision of a version history. These compound refactorings are called impure refactorings [8]. Two typical impure refactorings are described as follows:

**A refactoring with other refactorings** – After the refactoring which extracts a method from a class C using Extract Method, the method is moved to another class using Move Method refactoring because the method is no longer used by C.

**A refactoring with other corrections** – In the middle of the refactoring which decomposes a complex expression into plain expressions in the condition part of a if statement with Introduce Explaining Variable, a developer finds a mistake in the condition. As a result, the developer fixes the condition after the refactoring.

Impure refactorings are common and are widely observable in real software development processes [9].

Almost all previous refactoring detection methods identify a code fragment related to refactorings by comparing the characteristics of pre/post-modified programs. However, multiple modifications to the same revision may cause these characteristics to collide with the conditions to find refactorings. Furthermore, multiple refactorings sometimes create transient situations where code fragments cannot be found in the post-modified program.

Most existing approaches [5] manually describe the detection conditions regarding a combination of refactoring and other modifications such as Move Method with minor modification in the body of the method. However, the supported combinations are limited. Because the number of possible combinations grows exponentially with the number of supported refactorings, we need a technique to detect these combinations of refactorings without manually describing the detection conditions for the specific combinations.

The main contribution of this paper is to propose a technique for detecting combinations of refactorings without manually describing the combination-specific detection conditions. The above problems are caused by the loss of the intermediate states which providing detail refactoring histories during program modifications. As a solution against the problems, we estimate the intermediate states by actually refactoring the pre-modified version of the program. Since too many adaptable refactorings hamper the choice of the most appropriate one, this situation necessitates an effective selection of the refactorings applicable through the pre-modified to post-modified versions of the program. Our method is one candidate attempting to solve the difficulties to detect a combination of refactorings.

### 3. Proposed Approach

#### 3.1 Detecting Refactoring with Graph Search

We propose a novel technique for detecting the occurrences of refactoring by using a graph search commonly applied within the field of Artificial Intelligence [10].

Searches are formalized as states $N$, initial state $n_0 \in N$, final states $G \subseteq N$, operators $O$, and the function of calculating the cost on a path (simply, path cost function) $g : N \rightarrow \mathbb{R}^+$. State transitions by operators $V \subseteq N \times N$ will give an oriented graph of $(N, V)$. Figure 2 shows an outline of a graph search problem, where finding an appropriate path from the initial to the final state is the prime target. Search algorithms should find a solution minimizing the total path cost.

Finding refactoring performed between an old program $P_{old}$ and a new program $P_{new}$ is represented as the following search problem: states $N$ is a set of programs, initial state $n_0 = P_{old}$, targeting states $G = \{P_{new}(= n_m)\}$, operators $O$ is a set of refactoring operations, and path cost $g$ is the number of performed refactoring through $n_0$ to the current state.

The search encompasses the repetition of two steps: selecting and expanding. The selecting step determines the current state of the search, and the expanding step generates a new state by applying operators to the current state. If the search succeeds, i.e., it reaches the final state $P_{new}$ from the initial state $P_{old}$ with the sequence of the operation, we can regard the performed modifications as the obtained refactorings.

To efficiently find an appropriate path, some informed search algorithms are proposed. The A search is one of the promising algorithms performing a best-first search. The A search estimates the total cost for the path involving a state $n$ by the evaluation function $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic function which gives an estimated cost through
the state \( n \) to the final state. In the search, we determine a next state to be preferentially selected by the value of the evaluation function. By using heuristics, we can reduce both time and space complexities.

Inventive heuristic functions and the strategies for appropriately choosing refactoring operators to expanding new states make it an effective way to find refactorings. The differences between pre/post-modified programs play a key role in determining both the heuristic function and the strategy for extracting operators in our method.

3.2 Procedure of the Search

The procedure of the search is described as follows:

1. Initialization: \( n \leftarrow n_0 \), \( Queue \leftarrow \emptyset \).
2. Iteration: repeats the following steps within limited repeating times:
   a. Calculating differences: This finds a difference between \( n \) and \( n_m \). If the size of the difference becomes zero, the search then terminates and the adapted refactoring processes could be the output.
   b. Expanding: This derives candidates of refactorings \( o_1, ..., o_k \) based on the differences, and enqueues pairs of the state and each candidate \( (n, o_1), ..., (n, o_k) \) into \( Queue \). The elements in \( Queue \) are sorted by the evaluation function.
   c. Selecting: This obtains \( \langle n', o' \rangle \) by dequeuing the top of \( Queue \), which has the highest priority for selection. If \( Queue \) is empty, the search terminates.
   d. Applying refactoring: This generates the next state by applying the refactoring operation to be selected: \( n \leftarrow \text{refactor}(n', o') \). In the refactoring application, pre/post-conditions of the refactoring are checked. If any condition fails, the procedure reverts to \( 2-c \) for trying another candidate of refactoring.

Figure 2 shows an example of the search flow. First, by the differences between \( n_0 \) and \( n_m \), the candidates of refactorings \( o_1, o_2, \) and \( o_3 \) are enqueued to the priority queue. The operator having the highest priority is \( o_2 \). We then apply the refactoring \( o_2 \) to \( n_0 \), and obtain another version of the program \( n_1 \). In the second iteration, by the differences between \( n_1 \) and \( n_m \), we obtain \( o_4 \) and \( o_5 \). In this situation, the queue has four operators: \( o_1, o_3, o_4, \) and \( o_5 \). We then obtain \( n_1 \) by the best operator \( o_4 \). The search is performed by iterating the expansion, selection, and application of refactoring candidates.

In the following subsections we describe each part of the procedure in detail.

3.3 Difference Calculation

The structural differences allow us to identify the candidates of refactoring and provide a heuristic distance in our methodology. We have employed the differences of tree structures because the heuristics should treat structural changes and the similarities of programs. For example, the differences include the change of a field visibility from public to private suggest the possibility of Encapsulate Field. Traditional line-based program differences, which are available with Dtrf[11], are not adequate to deduce these possibilities. Additionally, a set of structural changes is a good measure for detecting similarities between programs.

We perform a tree-difference detection technique to two versions of a program represented as Abstract Syntax Trees (ASTs). The differences are represented as a sequence of Edit Scripts, which means an operation modifying an AST. Types of the Edit Scripts are as follows:

- \( \text{add}(\text{node}, \text{position}): \text{node} \) is added to \( \text{position} \).
- \( \text{remove}(\text{node}, \text{position}): \text{node} \) is removed from \( \text{position} \).
- \( \text{change}(\text{before}, \text{after}, \text{position}): \) a node positioned at \( \text{position} \) is updated from \( \text{before} \) to \( \text{after} \).

3.4 Deriving Refactoring Operations

Candidates of refactorings are derived by the calculated differences. To achieve this we represent each refactoring operation as a sequence of atomic modifications: e.g., \( \text{add} \) or \( \text{remove} \). A sequence of atomic modification from a refactoring \( \text{seq}(o) \) is described as the most typical modifications when only the refactoring \( o \) is observed at a revision. The sequences of supported refactorings are represented as follows:

Add Parameter
- \( \text{add}(	ext{SingleVariableDeclaration}, \text{Class}_A, \text{method}_1) \)

Encapsulate Field
- \( \text{change}(	ext{public, protected, Class}_A, \text{field}_1) \) or
- \( \text{change}(	ext{public, private, Class}_A, \text{field}_1), \text{add}(	ext{MethodDeclaration}, \text{Class}_A) \)

Extract Method
- \( \text{remove}(	ext{Block, Class}_A, \text{method}_1), \text{add}(	ext{MethodDeclaration, Class}_A) \)

Inline Method
- \( \text{remove}(	ext{MethodDeclaration}, \text{Class}_A), \text{add}(	ext{Block, Class}_A, \text{method}_1) \)

Move Method
- \( \text{remove}(	ext{MethodDeclaration, Class}_A, \text{method}_1), \text{add}(	ext{MethodDeclaration, Class}_B, \text{method}_2) \)

Remove Parameter
- \( \text{remove}(	ext{SingleVariableDeclaration, Class}_A, \text{method}_1) \)

Rename Method
- \( \text{change}(	ext{name}_1, \text{name}_2, \text{Class}_A, \text{method}_1) \)

Rename Field
- \( \text{change}(	ext{name}_1, \text{name}_2, \text{Class}_A, \text{field}_1) \)

Rename Variable
- \( \text{change}(	ext{name}_1, \text{name}_2, \text{Class}_A, \text{method}_1, \text{variable}_1) \)

By comparing these sequences with the calculated dif-
ferences we can derive the candidates of refactorings. Candidates are derived when a subsequence of the differences matches the subset of the modification sequence related to the refactoring. Examples of the derived refactorings based on differences are following:

- `add(MethodDeclaration, X): Extract Method to X`
- `add(Block, X): Inline Method to X`
- `remove(Block, X): Extract Method from X`
- `change(public, private, X): Encapsulate Field to X`

### 3.5 Establishing Priorities to Refactorings

In a classical search, we enqueue all of the states, which are obtained by applying every operator to be derived. However, the refactoring application may be a time-consuming process. Since many of them are not selected in the search because of the evaluation function, we do not necessarily have to apply all the operators.

In our methodology, we adopt a technique of lazy application of operations and obtain the extended version of the classical search procedure. We enqueue and dequeue pairs of a state and an operator. The refactoring operation is actually performed when the search finds the pair having the highest priority. The extended heuristic function evaluating the pair of a state and a refactoring operation is defined as follows: $h(n, o) = d(n)/\alpha(o)$. In the formula, a new parameter of refactoring operation $o$ is introduced. $d(n)$ is a heuristic distance based on the structural differences between $n$ and $n_m$ and $\alpha : O \rightarrow [0, 1]$ is a likelihood of the refactoring $o$, which becomes higher when the refactoring is supposed to be performed. When a refactoring $o$ has lower likelihood value, $h(n, o)$ is estimated higher than actual distance $d(n)$ for decreasing the priority of the candidate. In this way, $h(n, o)$ is an estimation of $d(n')$, where $n'$ is a new state generated by applying $o$ to $n$. The usage of this extended heuristic function reduces the number of application of refactoring operations and it then leads to reduce the computation cost of the search. By using this, we calculate $f(n, o) = g(n) + d(n)/\alpha(o)$ as an evaluation function for making an priority order of refactoring candidates.

The likelihood of refactoring candidates $\alpha(o)$ is determined by the rate of satisfied refactoring sequences as follows:

$$\alpha(o) = \frac{\#(m \in \text{seq}(o) | \exists m' \in \text{Diff}. m \text{ matches } m')}{\#(\text{seq}(o))}.$$  

`Diff` denotes the set of Edit Scripts representing the differences between $n$ and $n_m$. In concrete terms, we can determine $\alpha$ for some refactorings, e.g.:

**Extract Method** – One of the possibilities to be derived

Extract Method is the addition of a method which does not exist in the older version of the program. If the removed block also exists, we judge that both modifications are observed, i.e., $\alpha = 2/2 = 1$ or otherwise $\alpha = 1/2 = 0.5$. Through this, our approach determines that the former case has a higher possibility to be refactored by Extract Method than the latter case. Our heuristic search treats both cases with giving priorities.

**Rename Method** – Rename Method is derived when only the name of a method is modified. This means that the modification representing Rename Method is equal to one part of differences, so that a candidate is evaluated as the maximum likelihood, i.e., $\alpha(o) = 1$ whenever the differences include the change of a method name.

The heuristic distance between two states $d(n)$ is determined by the differences. The calculation is conducted in a simple way: we sum up the weight of each Edit Script in the differences. The weights $w$ are determined by the granularity of source code structure; higher level of syntactic elements in AST have a larger weight than lower level of elements. Performing coarse-grained refactorings prior to fine-grained refactorings may enhance the termination of the search. In order to preferentially select coarse-grained refactorings, we emphasize the effect of their coarse-grained refactorings by weighting editing operations related to the refactorings. For example, we use $w = 2$ in the cases of additions or deletions of fields, and $w = 3$ in that of methods. Total weighted cost is regarded as a heuristic distance: $d(n) = \sum w$. The overall information of the weights are shown in Table 1.

| AST Type of editing target          | $w$ |
|-------------------------------------|-----|
| TypeDeclaration (representing classes) | 5   |
| MethodDeclaration                  | 3   |
| FieldDeclaration, Block            | 2   |
| Types lower level than ExpressionStatement | 0   |
| Otherwise                           | 1   |

There are cases where two or more candidates have the highest priority. The candidates having the highest and the same priority are compared as follows. Firstly, we preferentially select the refactorings which greatly affect the program. For example, we select Rename Method rather than Remove Parameter. We consider that a refactoring for methods has more significant impact to the program than that for parameters because methods include parameters. Secondly, when both candidates are the same refactorings, we greedily select them. That is, candidates are evaluated by $d(n)$ for comparing just their future path costs.

### 3.6 Discussion

#### 3.6.1 Optimality

The proposed heuristic function is not admissible: i.e., the estimated heuristic distance sometimes may be smaller than the actual number of refactorings that need to be performed. It means that the solution resulting from the search might not be the shortest path, i.e., there exists a smaller set of refactorings satisfying between the initial and final state. We consider that the admissibility is not required in our objective. This is due to the fact that 1) significantly deteriorated
results are rare and 2) the size of the refactoring sequence is not closely related to the understandability of the program differences.

3.6.2 Cost of the Search

We consider the computational cost of the proposed search is potentially high because each iteration process includes difference calculation and performing refactoring, which are often heavyweight processes. The number of iteration is involved with at least 1) the size of differences and 2) the number of the kinds of refactoring operations to be detected. Large size of differences often consists of multiple refactoring operations. Moreover, the larger number of refactoring operations we support, the more candidates of refactoring to be deduced. The estimation and evaluation of the cost by applying our approach to real and large-scale software version archives are needed and we consider them as parts of our future work.

4. Automated Tool

We have implemented automated tools with Java, for enabling our methodology. They have the following functionalities:

- calculating the differences between two programs,
- deriving refactoring operations from the differences,
- evaluating the priority of refactoring operations and ordering them, and
- applying refactoring to generate new states of program.

They are not completely integrated in the current version; i.e., the user has to use each tool separately and performs the search manually. In principle the search could be executed automatically [10], so that the more useful implementation helps the developer to detect the refactorings more easily.

For the differences calculation we used a tool XMLDiff [12]. As a preprocessing, Java programs are converted into XML as a representation of ASTs by using Eclipse JDT [13]. In the XML representation, properties of an AST node are represented as child elements and that of non-AST node as attributes.

For the application of refactorings, we also used Eclipse [13]. By using Eclipse, we benefit as it automatically checks which pre/post-conditions of refactorings are to be performed.

We have implemented the derivation and evaluation rules for dealing with the following nine refactorings: Move Method, Encapsulate Field, Extract Method, Inline Method, Rename Method, Rename Field, Add Parameter, Remove Parameter, and Rename Variable in this granular order (prior refactorings are more coarse-grained). They are effective, well-known, and Eclipse-supported refactorings and it is the reason why we selected them. Supporting a refactoring requires the definition of the sequences of edit scripts $\text{seq}(o)$ along with the refactoring browser’s functionalities to check the pre-condition and to perform the refactoring to a program. Since we believe most refactorings defined by Eclipse IDE could be supported by our approach, extending our tool to support them is a possible future work.

5. Case Studies

To show the feasibility of our approach, we actually applied our technique to an existing version archive, which includes multiple occurrences of refactoring. We selected two evaluation targets: 1) REUSAR, the implementation of our approach and 2) Fowler’s example shown in Sect. 2.

5.1 Case 1: REUSAR

The reason we use REUSAR’s archive is that we had previously obtained the knowledge of what refactorings had actually been applied. The revision history of REUSAR is archived by Subversion. From the REUSAR’s archive, we extracted two versions of programs: revision 1796 ($=n_0$) and 1799 ($=n_m$). The refactorings that were actually performed between them are as follows:

**Rename Method** – This renames the method calculateDistance in the class DistanceCalculator to calcDistance.

**Remove Parameter** – This removes a calcDistance’s parameter. The type of the parameter is List<Diff>. Diff is a class representing the information of differences such as addition and deletion.

The overview and details of the search are shown in Fig. 3 and Table 2. The table describes the information covering all of the differences, the contents of the priority queue, and both the values of the evaluation and the heuristic functions for each iteration of the search.

In the first iteration, we obtain the three parts of differences between $n_0$ and $n_m$. The differences $\Delta_1$, $\Delta_2$, and $\Delta_3$ illustrate the change of a method name, the removal of a method parameter, and the deletion of a block in a method, respectively. These modifications are from Remove Parameter refactoring. Our tool derives three candidates of refactoring from the differences. The candidates are shown in the column “Operators in the queue” in Table 2, ordered by their priorities. Although $o_1$ and $o_2$ have the same evaluation value ($f(n_0, o_1) = f(n_0, o_2) = 4$), our method selected $o_1$, which renames a method calculateDistance to calcDistance.
for $n_0$, as the most likely to be applied. It is because $o_1$ (Rename Method) is bigger refactoring than $o_2$ (Remove Parameter). After the application of the refactoring, we obtained the new program $n_1$.

In the second iteration, from the differences between $n_1$ and $n_m$, we obtained the differences $\Delta_4$, the deletion of a parameter, and $\Delta_5$, a deletion of a block in a method. The differences also deduce two candidates of refactorings. They are shown in Table 2 along with the rest of candidates of the first iteration. Newly deduced refactorings are $o_4$ and $o_5$. Due to the likelihood values, the heuristic value for $(n_1, o_4)$ is smaller than that of $(n_1, o_5)$. As a result, the most appropriate operation $o_4$, Remove Parameter refactoring to $n_1$, is selected and applied for obtaining new program $n_2$. The reason why $o_4$ was selected, although its type and the evaluation value are the same as $o_2$’s, is that $d(n_1)$ is less than $d(n_0)$ and we then greedily select the states and operations closer to the final state.

In the third iteration, there are no differences between $n_2$ and $n_m$. Therefore, $n_2$ is the same program as $n_m$.

Consequently, we conclude that two applied refactorings $o_1$ and $o_4$ will be the modification from $n_0$ to $n_m$. They are a good solution representing the modifications actually performed. This means that our tool can detect a combination of refactorings $o_1$ and $o_4$ without preparing the combination-specific detection condition. As a comparison, the work by Weißgerber et al. [5] cannot detect the combination of refactorings even though it can detect each of Rename Method and Remove Parameter.

For the result of the search, derived refactorings include Extract Method refactorings ($o_3$ and $o_5$), which are not actually performed. They are derived from the modification related with parameters of the method, which involved with Remove Parameter refactoring. However, under the ordering of the priority-queue, they are not selected. It is caused by the low likelihood value $\alpha$, i.e., the differences do not contain an addition of newly method declarations. This situation indicates our heuristic-based approach and evaluation function can be effective. Since the technique used multiple criteria such as heuristic distance, likelihood, or granularity of refactorings, it indicates the effectiveness of our prioritizing strategy.

### 5.2 Case 2: Fowler’s Example

We have also applied our approach to the shrank Fowler’s example shown in Sect. 2. The performed refactorings are Extract Method to $Customer\#statement$ and Move Method to $Customer\#amountFor$.

The overview of the search is shown in Fig. 4. We have detected the correct two refactorings ($o_2$ and $o_7$; indicated by the bold arrows) by using our technique. The iteration process occurred thrice. The first iteration deduced six candidates of refactoring and selected Move Method from $Customer\#statement$ to $Rental\#amountFor$. However, it failed because of the pre-condition defined by Eclipse IDE. In the second iteration, the tool selected Extract Method from a block in $Customer\#statement$ to a new method. It succeeded and we obtained a new method named newMethod1 in $Customer$. Finally, in the third iteration, the tool deduced new six candidates and selected one of them. The selected refactorings is Move Method from the above extracted method to $Rental\#amountFor$, and it succeeded. Since the obtained state is completely the same as the final

Fig. 4 The search overview in Case Study 2.
state, the detection procedure terminated. It means that our tool can also detect a combination of Move Method and Extract Method refactorings by using the search.

6. Related Work

6.1 Metric Based Approach

Demeyer et al. proposed a metric-based technique for extracting the occurrences of refactorings [7]. The technique calculates the metric values of pre/post-modified programs, and detects the refactorings that are supposed to be performed by analyzing the changes of the values. What kinds of refactoring can be detected depends on the variation and quality of the measured metrics. Furthermore, multiple occurrences of refactoring have a negative impact on the metric changes: a refactoring, which increments a metric, and another refactoring, which decrements the metric, occur at once. The changes caused by non-refactoring modifications also lead to the increasing of false positives.

6.2 Pre/Post-Condition Based Approach

Extraction techniques based on pre/post-conditions are also proposed [6], [8]. These approaches preparedly describe the characteristics of the pre/post-refactored programs such as a pattern of source code changes, and match them with observable modifications between pre/post-modified programs. There also exists the difficulty in being able to detect multiple occurrences of refactoring. These approaches do not consider intermediate states of the performed modifications between two versions of the program.

Weißgerber et al. extend the method by Görg et al. [8] with a code clone analysis [5]. This method can extract refactorings related with the changes of names, types, and visibilities, even though they are performed with the other modifications at once. However, supported combinations of refactorings are somewhat limited. For example, a combination of related changes of a method signature and moving, e.g., move a method and add a parameter to the method, could not be detected.

6.3 Graph Transformation

A technique regarding a refactoring as a state transition is also proposed [14]. The key differences between their approach and ours are the model of programs and the adopted search algorithm. Firstly, they regard a program as a graph and a refactoring as a graph transformation. Their formal descriptions of refactorings make it easy to check pre/post-conditions of refactorings. However, it cannot be applied for detecting fine-grained refactorings such as Extract Method, because they use transformation rules on a structure-based graph. Our approach can define more fine-grained refactorings modifying the bodies of methods. Besides this, our approach differs as we have the ability to reuse well-maintained transformation rules and pre/post-conditions defined by mature refactoring browsers managed by open communities. Secondly, they use depth-first search against our best-first search. It is not rare that some refactorings are performed prior to other refactorings when a large refactoring is performed as a whole modification. In this case the refactoring performed at the beginning is not strongly applicable. However, the effectiveness of depth-first search strongly depends on the first choice.

6.4 Search-Based Software Evolution

Searches are used for detecting not past cases but future possibilities of refactorings. Some researches aim at knowledge-based software evolution for finding the opportunities of refactorings by heuristic-based search [15]–[17]. The evaluation function is the key issue representing the differences between their approaches and ours. For their search, they evaluate the quality of target programs for improving it. On the other hand, we evaluate the similarity of two programs for improving the understandability of the modifications to be performed.

7. Conclusion and Future Work

In this paper, we have proposed a novel methodology in order to detect the occurrences of refactoring with a graph search technique. In order to detect impure refactorings, our approach deals with intermediate states of program modifications between two versions of a program that has been committed into version archives. We model the detection as a best-first graph search problem. The usages of the differences between the programs for deriving and evaluating candidates of refactorings bring about the effectiveness of our approach.

We are considering the following issues as potential future work:

Detecting refactorings together with non-refactoring modifications. We believe our technique has a potential to detect not only a combination of refactorings, but also that of refactorings and non-refactoring modifications. Detecting them requires new criterion to terminate the search even in the case the current state does not reach to the final state. Additionally, a method for visualizing differences including both refactorings and non-refactoring modifications should also be discussed.

Deriving the sequence of editing operations from refactoring scripts. Refactoring candidates are derived by using the sequences of editing operations associated with a refactoring. We are considering the possibility that this sequence could be generated semi-automatically from existing refactoring scripts [18].

Cost estimation and optimization. We should acquire empirical evidences from large-scale case studies and clarify the effect of heuristics on the computational cost of the search. It could enable us to fix the evaluation function, and thresholds for reducing the computational cost of the search and for increasing the accuracy of detection results.
Integrating and extending the supporting tool. The supporting tool shown in this paper is under the prototyping, so the detection procedure is not integrated conveniently or seamlessly. In principle our approach could be applicable without an analysts’ hand work. A larger-scale case study would require a fully integrated supporting tool. Additionally, extending our tool to support new refactorings is also desired.

Case study on a larger scale. Our case studies in this paper are preliminary at scale. Further evaluation should apply our technique on a larger scale and real version archives.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments on the earlier version of this paper. This work was partly supported by JSPS Grant-in-Aid for Young Scientists (Start-up) 2180018.

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[2] W.F. Opdyke, Refactoring Object-Oriented Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.
[3] P. Cederqvist, “Version management with CVS.” Available at http://ximbiot.com/cvs/manual/
[4] B. Behlendorf, C.M. Pilato, G. Stein, K. Fogel, K. Hancock, and B. Collins-Sussman, “Subversion.” Available at http://subversion.tigris.org/
[5] P. Weiβgerber and S. Diehl, “Identifying refactorings from source-code changes,” Proc. 21st International Conference on Automated Software Engineering (ASE 2006), pp.231–240, 2006.
[6] D. Advani, Y. Hassoun, and S. Counsell, “Extracting refactoring trends from open-source software and a possible solution to the related refactoring’ conundrum,” Proc. 2006 ACM Symposium on Applied Computing (SAC 2006), pp.1713–1720, 2006.
[7] S. Demeyer, S. Ducasse, and O. Nierstrasz, “Finding refactorings via change metrics,” Proc. 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000), pp.166–177, 2000.
[8] C. Görg and P. Weiβgerber, “Detecting and visualizing refactorings from software archives,” Proc. 13th International Workshop on Program Comprehension (IWPC 2005), pp.205–214, 2005.
[9] E. Murphy-Hill, C. Parnin, and A.P. Black, “How we refactor, and how we know it,” Proc. 31st International Conference on Software Engineering (ICSE 2009), pp.287–297, 2009.
[10] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 2002.
[11] E.W. Myers, “An O(ND) difference algorithm and its variations,” Algorithmica, vol.1, no.1, pp.251–266, 1986.
[12] Microsoft Corporation, “Using the XML diff and patch tool in your applications,” available at http://msdn.microsoft.com/en-us/library/aa302294.aspx
[13] The Eclipse Foundation, “Eclipse.” Available at http://www.eclipse.org/
[14] J. Pérez and Y. Crespo, “Exploring a method to detect behaviour-preserving evolution using graph transformation,” Proc. 3rd International ERCIM Workshop on Software Evolution (Evol.’07), pp.114–122, 2007.
[15] M. Harman and J. Clark, “Metrics are fitness functions too,” Proc. 10th International Symposium on Software Metrics (METRICS 2004), pp.58–69, 2004.

Shinpei Hayashi received a B.Eng. degree in information engineering from Hokkaido University in 2004. He also received M.Eng. and Dr.Eng. degrees in computer science from Tokyo Institute of Technology in 2006 and 2008, respectively. He is currently an assistant professor of computer science at Tokyo Institute of Technology. His research interests include software evolution and refactoring, software patterns, software development environment, and mining software repositories.

Yasuyuki Tsuda received B.Eng. and M.Eng. degrees in computer science from Tokyo Institute of Technology in 2005 and 2007, respectively. He is currently working at Yokogawa Electric Corporation. His research interests include software development environment, software refactoring, software reuse, and reverse engineering techniques.

Motoshi Saeki received a B.Eng. degree in electrical and electronic engineering, and M.Eng. and Dr.Eng. degrees in computer science from Tokyo Institute of Technology, in 1978, 1980, and 1983, respectively. He is currently a professor of computer science at Tokyo Institute of Technology. His research interests include requirements engineering, software design methods, software process modeling, and computer supported cooperative work (CSCW).