Searching for Opportunities of Refactoring Sequences: Reducing the Search Space

Eduardo Piveta¹, João Araújo², Marcelo Pimenta¹, Ana Moreira², Pedro Guerreiro³, R. Tom Price¹

¹Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil {epiveta, mpimenta}@inf.ufrgs.br, tomprice@terra.com.br
²Departamento de Informática, FCT, Universidade Nova de Lisboa, Caparica, Portugal {amm, ja}@di.fct.unl.br
³Departamento de Engenharia Electrónica e Informática, Universidade do Algarve, Faro, Portugal pjguerreiro@ualg.pt

Abstract

During software development and evolution activities, the developers focus the refactoring efforts on choosing and applying refactoring patterns (or sequences of patterns) that are likely to improve the software quality. Considering the search for opportunities of refactoring sequences, the main problem is the size of the search space (there are too many possible sequences to be evaluated). We propose an approach to narrow the number of refactoring sequences by discarding those that semantically does not make sense and avoiding those that lead to the same results. We provide a detailed example of the approach considering sequences for method manipulation, showing how the number of sequences can be significantly reduced.

1 Introduction

Refactoring [12, 5, 11] is the process of improving the design of software systems without changing their externally observable behaviour. Refactoring can help to incrementally improve the quality attributes [8, 2] of a software system through the application of behavioural preserving transformations called refactoring patterns.

As these refactoring patterns are usually low grained transformations, the application of an individual refactoring pattern may not be enough to bring substantial benefits to the software quality attributes’ satisfaction, but it can enable the application of other patterns [16]. When defining refactoring sequences (the application of refactoring patterns in sequence), it is useful to determine which patterns can be applied in parallel and which cannot [13].

Current research on the identification of refactoring opportunities [4, 10, 3] focuses on improvements considering the individual application of a refactoring pattern, but it does not consider how this identification can be conducted in the context of sequences of refactoring.

Even when we use only a few refactoring patterns, the number of possible sequences is very large. Consequently, it is important to answer the following questions:

(a) How can we narrow the refactoring sequences to those that are semantically sound and avoiding sequences leading to the same results?

(b) And then, from these remaining sequences, how can we know which are the best sequences (according to the quality attributes satisfiability)?

The goal of this paper is to answer the first question, by proposing an approach to reduce the number of possible refactoring sequences, including the simplification of equivalent, commutative, inverse, forbidden and parallel sequences. This simplification can help the developer to focus on searching for refactoring opportunities for the most promising refactoring sequences.

We exemplify the approach through the use of Deterministic Finite Automata (DFA) [15] to represent refactoring sequences for method manipulation and a set of simplification rules to reduce the search space. We show that the number of sequences can be greatly reduced by simplification (in the example, the search space showed a 62% reduction on the number of sequences).

This paper is organised as follows. Section 2 presents the motivation for the simplification of possible refactoring se-
quences. Section 3 describes how to create an initial representation of the possible sequences and how to simplify this representation considering a set of simplification rules. Section 4 shows how the approach can be used in practice using a set of refactoring patterns to manipulate methods. Section 5 discusses issues regarding the search for refactoring opportunities considering refactoring sequences. Section 6 describes related work comparing it with our approach. Section 7 includes some concluding remarks and directions for future work.

2 Motivation

The number of places where it is possible to refactor can be large. Therefore, when searching for refactoring opportunities, we need to reduce the search space either by reducing the number of refactoring patterns or by imposing constraints for each refactoring pattern.

Consider, for example, the possible places for four different refactoring patterns: Pull Up Method, Extract Interface, Rename Class and Inline Method.

The number of targets for refactoring can be counted as:

- **Pull Up Method**: For each class that has a super-class, all the methods can be targets. Let \( C_c \) be the set of classes with a super-class and let \( m(x) \) be a function that, given a set of classes returns a set of methods defined on these classes, the number of possible applications of this refactoring pattern is \( |m(C_c)| \).

- **Extract Interface**: For each class, several different interfaces can be created: an empty interface, an interface with one method (any method of the class), with two methods, etc. Considering \( C \) as the set of classes in a given system, and \( n(c) \) as the number of methods of a class \( c \), the number of possible applications of this refactoring pattern in a class \( c \) can be computed by the \( ei(c) \) function:

  \[
  ei(c) = \sum_{i=0}^{n(c)} \frac{n(c)!}{i!(n(c) - i)!}
  \]

- **Rename Class**: All classes can be renamed so, there is one target per class. In this case, the number of possible applications is \(|C|\), where \( C \) is the set of classes in a given system. Note that, in this case, the variability in the parameter values (i.e. the new name) is not taken into account.

- **Inline Method**: For each method, in each class, there is one possible target for refactoring. Using the \( m(x) \) function, the number of possible applications of this pattern is given by \(|m(C)|\).

Therefore, the number of possible applications of the selected refactoring patterns can be computed by:

\[
P = |m(C_c)| + \sum_{i=1}^{\left\lfloor \frac{|C|}{m(C)} \right\rfloor} ei(c_i) + |C| + |m(C)|
\]

Considering a simple system, with ten classes \((c_1, \ldots, c_{10})\), with ten methods in each class, where the first five classes are sub-classes, the number of possible refactoring application is:

\[
C = \{c_1 \ldots c_{10}\}
\]

\[
Cc = \{c_1 \ldots c_3\}
\]

\[
P = m(C_c) + \sum_{i=1}^{\left\lfloor \frac{|C|}{m(C)} \right\rfloor} ei(c_i) + 5 + m(C)
\]

\[
P = 50 + 1024 + 5 + 100 = 1179
\]

This confirms that, even for a small system, the number of possible places to refactor is large. Considering the application of refactoring sequences, the number can greatly increase, as for each individual application, the developer can search for additional refactoring patterns that can be applied to the software.

A concrete combination of these sequences is computed by:

\[
P = \binom{n}{k}
\]

where \( n \) is the number of possible applications of the refactoring patterns and \( k \) is the sequence size.

In the example, it is impractical to evaluate all possible sequences (\( \binom{10}{3} \)). The developer must define additional constraints to further reduce the search space. The set of refactoring opportunities can be reduced using two different strategies:

a **Reducing the number of possible combinations of the refactoring patterns.** This is the focus of this paper. Section 3 describes an approach to reduce the number of possible refactoring patterns and Section 4 exemplifies the approach for a set of refactoring patterns.

b **Reducing the parameters to be passed to the refactoring patterns.** For example, the developer can search for refactoring opportunities of Extract Interface with zero, 50% and 100% of the methods, to reduce the total number of targets (in this case, reducing from 1024 to 254 opportunities). This strategy is not addressed in this paper but it will be handled in future research.
3 Reducing the Search Space

This section describes the steps needed to reduce the number of refactoring sequences to be evaluated by the developer. To achieve that, two activities must be performed:

- **Create the initial refactoring sequences**: An initial representation of all the possible sequences of refactoring is created, regardless of the semantics of each refactoring pattern. This representation can be expressed using graphs, finite state machines, Petri nets, etc.

- **Simplify the set of sequences**: the initial representation is simplified, considering the semantics of each transformation. In this step, the created representation is traversed, searching for simplifications or equivalences between different sequences, commutative, inverse, and for independent or forbidden sequences.

Figure 1 shows the workers and work products for these two activities (according to the SPEM notation [6]). The tool provider creates the initial representation for the refactoring sequences and then simplifies it using a set of rules. A developer can then use this simplified representation to search for refactoring opportunities for these sequences (or for a sub-set of these sequences).

![Diagram of refactoring process](image)

**Figure 1. Workers, Work Products and Activities**

Section 3.1 describes the creation of this initial representation using a deterministic finite automaton and Section 3.2 describes the simplification rules and how each simplification can be done.

3.1 Creating the Initial Refactoring Sequences

The first step is to create a representation of all the refactoring sequences for each element of the grammar. Given a refactoring catalog and the language grammar, a set of representations can be created, one for each grammar element. The procedure works as follows. For each element in the grammar, the refactoring patterns in the catalog are visited. If the refactoring pattern is applicable to the current element, it is added to the representation of the element if it already exists; if not, a representation of the sequences is created for the element.

A more detailed algorithm to the creation of the representation of the sequences can be defined as follows, where `Catalog` is a refactoring catalog, `Grammar` is a language grammar and `Levels` is the number of levels for the created representation:

![Algorithm for creating refactoring sequences](algorithm)

**CREATE-REPS(Catalog, Grammar, Levels)**

1. \( A \leftarrow \emptyset \)
2. **for each** \( e \) **in** Grammar
3. **do for each** \( l \) **in** Levels
4. **do for each** \( rp \) **in** Catalog
5. **if** \( rp.ISAPPLICABLETO(e) \)
6. **then**
7. \( if \ rep(e) \notin A \)
8. **then** \( A \leftarrow A \cup NREP(rp, l); \)
9. **return** \( A \)

The `isApplicableTo` method checks whether the refactoring pattern \( rp \) is applicable to the element \( e \). Function `nREP` creates a representation for a given element and the `add` method adds a refactoring pattern to all the states in a given representation that are in a given level. To simplify the creation process, the forbidden paths are not evaluated at this stage, but later, in the simplification phase. A concrete example is shown in Section 4.

3.2 Simplifying the Sequences

The next steps reduce the number of sequences. Let \( r1 \) and \( r2 \) be refactoring patterns and \( I \) be an initial state. The following situations can occur:

- **Simplifications**: Simplifications occur when there is a shorter path that leads from an initial state to an end state. If from \( I \), the application of \( r1 \) followed by \( r2 \) results in a shorter path that leads from an initial state to an end state. The sequences are said to be equivalent. This equivalence can be denoted as: \( I \ r1 \ r2 = I \). For example, `Pull Up Method` is the inverse of `Push Down Method`. This case can be expressed as: \( I \ r1 \ r2 = I \). It is a special kind of simplification.

- **Commutative Path**: Commutative paths occur when the order of application of a refactoring patterns pair does not matter. It means that: \( I \ r1 \ r2 = I \ r2 \ r1 \).

- **Inverse Path**: Refactoring patterns usually have an inverse refactoring pattern (for example, `Pull Up Method` is the inverse of `Push Down Method`). This case can be expressed as: \( I \ r1 \ r2 = I \). It is a special kind of simplification.
Table 1. Selected Refactoring Patterns

<table>
<thead>
<tr>
<th>Ref. Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull Up Method</td>
<td>The method is moved to the superclass. All calls to the method are replaced by the contents of the method. The method is deleted.</td>
</tr>
<tr>
<td>Inline Method</td>
<td>The method is renamed as well as all calls to it.</td>
</tr>
<tr>
<td>Rename Method</td>
<td>The method is moved to another class and the references are updated.</td>
</tr>
<tr>
<td>Move Method</td>
<td>The method is moved to one or more subclasses of the class containing the method.</td>
</tr>
</tbody>
</table>

- **Independent Path**: This kind of sequence occurs when two different refactoring patterns in a path can be applied in parallel (i.e. the refactoring patterns do not have an influence on each other). It is a special case of a commutative path.

- **Forbidden Paths**: Forbidden paths are refactoring sequences that cannot be applied. Certain refactoring patterns can disable the application of other patterns. For example, after a method is inlined it cannot be moved or renamed because the method itself does not exist anymore.

These rules of behavioural preservation and equivalence can be proved using different techniques. The equivalences for simplifications, commutative and inverse paths can be proved using graph parallelism and confluence techniques [1, 7]. The occurrence of independent and forbidden paths can be detected using critical pair analysis [10].

The following algorithm can be used to simplify a set of sequences for a given element of the grammar. First, all possible sequences are computed. Then, the forbidden, independent and inverse sequences are removed, simplifications are removed and finally, one of the commutative sequences are also removed (it does not matter which one). The algorithm can be expressed as:

```
SIMPLIFY-REP(rep)
1 seqs = GETSEQS(rep)
2 seqs = seqs − FORBIDDENSEQ(SEQS)
3 seqs = seqs − INDEPENDENTSEQ(SEQS)
4 seqs = seqs − INVERSESEQ(SEQS)
5 seqs = seqs − SIMPLIFICATIONSEQ(SEQS)
6 seqs = seqs − COMMUTATIVESEQ(SEQS)
7 return seqs
```

Functions *ForbiddenSeq*, *IndependentSeq*, *InverseSeq*, *SimplificationSeq* and *CommutativeSeq* return, respectively, all the sequences that are forbidden, independent, inverse, simplifications and one of two commutative paths. A concrete example using these rules is shown in Section 4.

### 4 Example: Refactoring Methods

In this section, deterministic finite automata (DFAs) are used to represent the possible refactoring sequences for a set of refactoring patterns. DFAs are a practical model of computation [15], and there are several algorithms to find a DFA recognizing the union, intersection, and complements of the recognized languages.

An initial DFA is simplified through the application of the rules defined in the previous section to exemplify the approach. The simplification of the DFA for the Pull Up Method paths is explained in detail and for the other remaining paths is described more briefly. The initial and final DFAs are compared, showing the differences before and after the simplification is performed.

Table 4 shows five refactoring patterns for the manipulating methods in a class were selected for the purpose of this example.

#### 4.1 Creating the initial DFA and Removing Forbidden Sequences

The first step is to create a DFA describing the sequences of refactoring patterns for method manipulation. The following representations are used to express each refactoring pattern:

- \( \text{pu}(m,s) = \text{pullUp} \) (method, superClass)
- \( \text{im}(m) = \text{inline} \) (method)
- \( \text{rm}(m,n) = \text{rename} \) (method, newName)
- \( \text{mm}(m,nc) = \text{move} \) (method, newClass)
- \( \text{pd}(m,sc) = \text{pushDown} \) (method, subClass)

As we have observed, the Inline Method refactoring stops the possibility of manipulating the method. Therefore, the first step is to remove the forbidden paths thus generated. Figure 2 shows the initial DFA with the forbidden paths removed.

Note that even though the refactoring patterns can be applied, this does not imply that it is sound or beneficial to apply them. From the initial state \( s_1 \), 21 different paths can be followed.
4.2 Simplifying the Pull Up Method Sequences

In the second step, after the application of a Pull Up refactoring pattern, other patterns can be applied in sequence, such as: another Pull Up Method, an Inline Method, a Rename Method, a Move Method or a Push Down Method. We start with this particular branch of the initial DFA (Figure 2). Figure 3 shows the branch of Pull Up Method considering the application of two patterns in sequence. This Pull Up Method branch will be simplified by applying a set of rules.

4.2.1 Simplification: Pull Up – Pull Up Sequences

The DFA is traversed, starting from the first sequence (Pull Up followed by another Pull Up). In practice, this first sequence is unnecessary, as the developer can pull up the method two classes up in the inheritance tree. There is no need to look for opportunities to first pull up to the immediate superclass and then look for an opportunity to pull up one class more in the hierarchy tree. This led us to the first simplification rule in the refactoring DFA.

Whenever we have a sequence of two applications of the Pull Up Method refactoring pattern involving the same method, the path can be simplified in a way that the method is pulled up to the higher superclass in the inheritance tree (the superclass in the second Pull Up Method occurrence). This rule can be expressed as follows:

\[ I.\text{pu}(m, s) \cdot \text{pu}(m, s') = I.\text{pu}(m, s') \]  

(1)

Therefore, in the initial DFA, the Pull Up – Pull Up sequence can be removed: \(pu(m, s) \cdot pu(m, s')\)

4.2.2 Simplification: \(\lambda – \text{Inline Sequences}\)

The second path that can be simplified occurs when a Pull Up Method is followed by an Inline Method. The application of Inline Method replaces all the calls to the method by the method’s contents. Moving the method to the superclass and then inlining it in the same refactoring sequence does not make sense, and therefore we will reject that possibility. This simplification can be expressed as follows:

\[ I.\text{pu}(m, s) \cdot \text{im}(m) = I.\text{im}(m) \]  

(2)

In fact, after applying the Inline Method, the application of the remaining selected refactoring patterns in Table 4 does not make sense. Therefore, the sequences can be simplified to the direct application of Inline Method, instead of applying first the other refactoring patterns.

In summary, whenever a sequence contains a Pull Up Method, a Rename Method, a Move Method or a Push Down Method refactoring pattern followed by an Inline Method operating in the same method, the sequence can be simplified in a way that only the Inline Method refactoring pattern is applied. This rule can be expressed as follows:

\[ I.\lambda \cdot \text{im}(m) = I.\text{im}(m) \]  

(3)

where \(\lambda \in \{\text{pu}(m, s), \text{rm}(m, n), \text{mm}(m, nc), \text{pd}(m, sc)\}\)

This simplification leads to the removal of all the sequences ending with an Inline Method: \(\text{rm}(m, n) \cdot \text{im}(m)\), \(\text{mm}(m, nc) \cdot \text{im}(m)\), \(\text{pu}(m, s) \cdot \text{im}(m)\) and \(\text{pd}(m, sc) \cdot \text{im}(m)\).
4.2.3 Commutative Paths: Rename and Pull Up Sequences

Another common simplification is the occurrence of commutative paths. By observing the sequence Pull Up Method followed by the application of Rename Method in a different order, we conclude that it does not matter if the method is renamed before or after the Pull Up Method application. This leads us to the next simplification rule, which can be summarized as follows:

If there is a sequence composed by a Pull Up Method followed by a Rename Method, the inverse sequence (Rename followed by Pull Up) can be removed from the DFA (or vice-versa). This rule can be expressed as:

\[ I.pu(m, s).rm(n, m) = I.rm(n, m).pu(m, s) \]  
(4)

Figure 4 shows the rule expressed as a DFA.

In this case, we arbitrarily choose to remove the Rename Method – Pull Up Method sequence: \( rm(m, n).pu(m, s) \). It does not matter which one of the commutative paths is chosen for removal.

4.2.4 Simplification: Pull Up – Move Sequences

The next simplification refers to the situation when a sequence composed of a Pull Up Method followed by a Move Method operate in the same method. The Pull Up Method does not change the overall result, as the method is moved again to another class. This led us to another simplification rule, stating that whenever a Pull Up Method is applied before a Move Method operating in the same method, the sequence can be simplified to the initial state followed by the application of Move Method. The definition is as follows:

\[ I.pu(m, s).mm(m, nc) = I.mm(m, nc) \]  
(5)

In this case, the larger sequence \( pu(m, s).mm(m, nc) \) is removed from the initial DFA.

4.2.5 Inverse: Pull Up – Push Down Sequences

One additional simplification that can be performed is the search for inverse paths, i.e. paths that reverse the effects of a previously applied refactoring. For example, if a method is pulled up from a class to a superclass and after that it is pushed down to the original class again, all the classes remain the same. Therefore, whenever a sequence of a Pull Up Method followed by a Push Down Method is applied to the same method in the same classes, the end result is the initial state:

\[ I.pu(A.m, B).pd(B.m, A) = I \]  
(6)

This situation does not occur when the classes involved in the operations are different. For example, if a method is pulled up from class A to class B and after that it is pushed down from class B to class C, the resulting state is different from the original state:

\[ I.pu(A.m, B).pd(B.m, C)! = I \]  
(7)

In this case, the initial \( pu(m, s) . pd(m, sc) \) sequence stays in the DFA, until the concrete parameters are evaluated to see if it is an inverse sequence.

4.3 Comparing the DFAs

In summary, the Pull Up Method branch of the DFA was simplified with the following rules:

\[
\begin{align*}
I.pu(m, s).pu(m, s') &= I.pu(m, s') \\
I.pu(m, s).im(m) &= I.im(m) \\
I.pu(m, s).mm(m, nc) &= I.mm(m, nc) \\
I.pu(m, s).rm(m, n) &= I.rm(m, n).pu(m, s) \\
I.pu(A.m, B).pd(B.m, A) &= I
\end{align*}
\]

Repeating the same approach for the Rename Method, Move Method and Push Down Method branches, the DFA is further simplified. The following additional rules were applied to simplify the DFA (for the sake of space, we do not discuss these simplifications in detail here):

\[
\begin{align*}
I.rm(m, n).rm(n, m') &= I.rm(m, n') \\
I.rm(m, n).mm(m, nc) &= I.mm(m, nc).rm(n, m) \\
I.mm(m, nc).mm(m, nc') &= I.mm(m, nc') \\
I.pd(m, sc).pu(m, s') &= I \\
I.pd(m, sc).mm(m, nc) &= I.mm(m, nc) \\
I.pd(m, sc).pd(m, sc') &= I.pd(m, sc')
\end{align*}
\]
Figure 2 shows the initial DFA and Figure 5 shows the simplified DFA. We achieved a reduction of 62% of the initial number of sequences. Note that when creating the DFA with the application of the refactoring patterns to real software projects, the DFA is larger, as there is a path for each possible application of each pattern defined in the initial DFA.

![Simplified DFA with 8 paths](image)

The third question, which is the focus of this paper, includes the reduction of the refactoring opportunities search space, by creating refactoring sequences that can be later analysed and applied if desirable. The successive application of refactoring patterns can be explored to search refactoring opportunities for the application of two or more refactoring at once. For example, instead of searching only for the application of a Pull Up Method refactoring pattern, the developer can also search for opportunities that are created with the new method in the super classes.

The search for refactoring opportunities can be extended to sequences of two or more refactoring patterns. The developer can then focus on:

1. Searching for refactoring opportunities for all refactoring patterns in a catalog.
2. Searching for refactoring opportunities for the $X$ best ranked refactoring patterns (according to a previously defined ranking), considering sequences of $n$ refactoring patterns.
   
   (a) Also, search for refactoring opportunities that enable the application of the $X$ best ranked refactoring patterns, considering sequences of $n$ refactoring patterns. These enabling refactoring patterns are those that do not improve directly the selected quality attributes but creates a refactoring opportunity for a refactoring pattern that did not exist before the application of the enabling refactoring pattern.
   
   (b) Also, take into account refactoring opportunities that disable the application of the $X$ best ranked refactoring patterns, considering sequences of $n$ refactoring patterns. Critical pair analysis [10] can be used to spot refactoring patterns that disable the application of others.

Nevertheless, the evaluation of the effects of refactoring sequences on quality attributes and the reduction of the possible parameters for each refactoring pattern is still an open question and can be the subject of further research.

6 Related Work

Mens et al [9] describe current trends and future research regarding refactoring in general. They observe that determining where and why refactoring patterns should be applied is still an open problem. Here, we presented an approach to reduce the number of places to evaluate when searching for refactoring opportunities of refactoring sequences.

Tourwe and Mens [16] use logic meta-programming to search for refactoring opportunities (including sequences of
refactoring) in existing software. They state that identifying opportunities for refactoring sequences requires checking opportunities for each and every possible refactoring, which could take quite too much time and should be the focus of future work. We extend their work by providing mechanisms to reduce the search space, reducing the effort needed to search for opportunities for refactoring sequences.

Mens et al. [10] also explore the problem of structural evolution conflicts in a formal way by using graph transformation and critical pair analysis. They show how this formalism can be used to detect and resolve refactoring conflicts. Heckel et al. [7] establish a definition of critical pairs for typed attributed graph transformation and provide a critical pair lemma. According to them, local confluence follows from confluence of all critical pairs. Their techniques are used by our approach to help detecting forbidden sequences and independent sequences.

Approaches for identifying refactoring opportunities [14] and for evaluating the effects of refactoring on quality attributes [3, 4] can be adapted to be used together with our approach, in order to identify refactoring opportunities by considering refactoring sequences.

7 Conclusion

This paper described an approach for reducing the search space for refactoring opportunities, by providing mechanisms to create and simplify a DFA representing the applicable refactoring sequences in existing software.

The approach was exemplified using five refactoring patterns dealing with the manipulation of methods. The initial DFA was simplified and its size was reduced in 62% (considering the total number of paths to be evaluated).

Additional techniques can be used to further reduce the scope of refactoring, including the careful selection of the refactoring patterns to be included in the search, the modules to be evaluated and the optimal parameters for each refactoring pattern.

Future work should focus on answering the question of which are the best sequences (according to the quality attributes satisfiability) and on further techniques to reduce the search space, more specifically in the task of choosing the right parameters for the concrete application of the refactoring patterns being used.

References


