JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2012; **00**:1–31
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Understanding Systematic and Collaborative Code Changes by Mining Evolutionary Trajectory Patterns

Qingtao Jiang[1,2], Xin Peng[1,2], Hai Wang[1,2], Zhenchang Xing[3], Wenyun Zhao[1,2]

[1]*School of Computer Science, Fudan University, Shanghai, China*
[2]*Shanghai Key Laboratory of Data Science, Fudan University, China*
[3]*School of Computer Engineering, Nanyang Technological University, Singapore*

SUMMARY

The lifecycle of a large-scale software system can undergo many releases. Each release often involves hundreds or thousands of revisions committed by many developers over time. Many code changes are made in a systematic and collaborative way. However, such systematic and collaborative code changes are often undocumented and hidden in the evolution history of a software system. It is desirable to recover commonalities and associations among dispersed code changes in the evolutionary trajectory of a software system. In this paper, we present SETGA (Summarizing Evolutionary Trajectory by Grouping and Aggregation), an approach to summarizing historical commit records as trajectory patterns by grouping and aggregating relevant code changes committed over time. SETGA extracts change operations from a series of commit records from version control systems. It then groups extracted change operations by their common properties from different dimensions such as change operation types, developers and change locations. After that, SETGA aggregates relevant change operation groups by mining various associations among them. We implement SETGA and conduct an empirical study with three open-source systems. We investigate underlying evolution rules and problems that can be revealed by the identified patterns and analyze the evolution of trajectory patterns in different periods. The results show that SETGA can identify various types of trajectory patterns that are useful for software evolution management and quality assurance.

Received ...

KEY WORDS: Evolution, Version Control System, Code Change, Pattern, Mining.

1. INTRODUCTION

A large-scale software system can undergo many releases in its life cycle. Each release often involves hundreds or thousands of revisions committed by many developers over time. To obtain a high-level overview of software evolution, developers often need to manually inspect a large amount of revisions by examining program differences and log messages. This manual process is tedious and time-consuming. Developers can easily get lost due to overwhelming details. Although change logs sometimes provide brief descriptions of the changes, they can only cover the changes in individual commits while closely related changes often span multiple commits. Moreover, change logs usually lack important information such as the developers, positions (e.g., classes/methods), and orders of involved changes.

*Correspondence to: Xin Peng, School of Computer Science, Fudan University, Shanghai, China. Email: pengxin@fudan.edu.cn

Many code changes are made in a systematic and collaborative way. Individual code changes are often an integral part of high-level change requests such as fixing bugs, introducing new features, enhancing existing features, and refactoring. Such high-level change requests often involve a group of related code changes in multiple places to ensure consistency and completeness [11]. Furthermore, a series of relevant code changes are often made according to explicit or implicit schedules reflecting specific development processes and collaboration modes.

As an example, consider an online shopping system that implements only one payment mode currently. To support two new payment modes, a developer, Jack, extracts several code fragments from existing methods that are common to all payment modes as separate methods. After that, he adds method calls from the original methods to the extracted common methods. Another developer, Tom, reuses the extracted common methods to implement the two new payment modes. All the above code changes are made for a high-level change request, i.e., supporting two new payment modes. The change process reflects the development process of refactoring existing implementation first and then introducing new functionalities. The changes also reflect the collaboration between Jack and Tom.

To obtain a high-level overview of the evolution history of a software system, it is desirable to summarize historical change records as high-level change patterns that can reveal commonalities and associations of dispersed code changes. Researchers have used program differencing and data mining techniques to identify co-change patterns [19, 21, 3] and fine-grained repetitive code changes [14, 13]. These approaches can only relate code changes within the same transactions or time windows. There also have been some approaches [10, 9, 11] that can group systematic code changes as logic rules. However, these approaches do not consider associations among different groups of changes or the time and developer properties of individual code changes. As such, they cannot reveal patterns of evolutionary trajectory.

In this paper, we propose the concept of trajectory patterns and the SETGA (Summarizing Evolutionary Trajectory by Grouping and Aggregation) approach that can summarize historical change records as trajectory patterns by grouping and aggregating relevant code changes committed over time. SETGA takes as input a series of commit records of a software system from version control systems. It first extracts change operations from each commit by comparing source code of involved files before and after the revision. The extracted change operations are then grouped by their common properties of different aspects such as change operation types, developers, and locations of the changes. After that, SETGA aggregates relevant groups of change operations together by mining various associations among them such as method calls, field accesses and similar change content.

We have implemented SETGA[†] and conducted an empirical study with three open-source systems. We categorized the identified trajectory patterns, evaluated the span of time, space and developers of code changes involved in trajectory patterns, and analyzed the impact of different thresholds on the identification of trajectory patterns. The results demonstrate the usefulness and effectiveness of SETGA for summarizing evolutionary trajectory for software maintenance. We further investigated underlying evolution rules and problems that can be revealed by the identified patterns and analyzed the evolution of trajectory patterns in different periods.

The rest of the paper is structured as follows. Section 2 reviews some existing proposals and compares them with ours. Section 3 defines trajectory pattern and other related concepts. Section 4 describes the proposed approach. Section 5 presents the empirical study with three open-source systems. Section 6 discusses some related issues. Section 7 concludes the paper and outlines the future work.

[†]SETGA Tool is available at: http://www.se.fudan.edu.cn/research/TrajectoryPattern

DOI: 10.1002/smr

2. RELATED WORK

One of the main purposes of mining software repositories (MSR) is to analyze trends and recurring patterns of software changes. Kagdi et al. [8] presented a comprehensive literature survey on approaches for mining software repositories in the context of software evolution.

Some approaches aim to analyze evolution phases and styles of software evolution. Xing and Stroulia [18] proposed an approach for analyzing the evolution history of the logical design of object-oriented software systems. The approach recovers a high-level abstraction of distinct evolutionary phases and their corresponding styles and identifies class clusters with similar evolution trajectories.

Visualization has been used to study the evolution history of software systems. Gall et al. [6] presented a three-dimensional visual representation for examining a system's release history. Collberg et al. [4] presented a visualization system that extracts evolution information from version control systems and displays it using a temporal graph visualizer. Van Rysselberghe and Demeyer [17] applied a simple visualization technique to recognize relevant changes such as unstable components and coherent entities. Beyer and Hassan [2] proposed a visualization technique that automatically extracts software dependency graphs from version control systems and computes storyboards based on panels for different time periods. D'Ambros et al. [5] proposed a visualization-based approach that provides module-level and file-level co-change information. The approach supports the analysis of evolution coupling over time.

There has been some work on mining code change patterns from version control systems. Ying et al. [19] applied data mining techniques on the revision history of a system to detect source files that are usually changed together. Zimmermann et al. [21] proposed an approach that mines association rules among the changes of program entities such as functions or variables. Bouktif et al. [3] proposed an approach that uses a pattern recognition technique to discover co-change patterns among files. These approaches can only relate changes that occur in the same transactions.

Some recent work has been focused on fine-grained code changes. Nguyen et al. [14] extracted method-level code changes by comparing abstract syntax trees (ASTs) of two consecutive revisions. They studied within- and cross-project repetitiveness of code changes with a large data set. A recent work by Negara et al. [13] proposed an approach that mines frequent code change patterns from a fine-grained sequence of change operations to AST nodes captured by IDE-based event trackers. They used a time window mechanism to discern the boundaries between transactions to mine a continuous sequence of code changes ordered by timestamp. Therefore, their approach can only mine fine-grained change patterns that are smaller than a given time window (e.g., five minutes).

All in all, trajectory patterns identify high-level changes that span a series of commits and are made systematically in different places and collaboratively by different developers. Kim et al. [10, 9, 11] proposed a rule-based program differencing approach that automatically discovers and summarizes systematic code changes as logic rules. Their approach captures change rules at two different abstraction levels, i.e., changes to method-header names and signatures, changes to code elements and structural dependencies. The rule inference in their approach corresponds to the grouping of change operations in our approach. Based on the identified change operation groups, our approach further aggregates relevant groups according to various associations. Moreover, their approach infers change rules from two program versions, while our approach analyzes historical commit records during a given period. This enables us to identify trajectory patterns that reflect intermediate process such as time sequences, developers of commit records, and transient code changes.

Our earlier work [7] presents basic concepts and algorithms of trajectory patterns and reports a preliminary empirical study with three open-source systems. This paper extends the work by providing more detailed descriptions of the algorithms and an extended empirical study. In the study, we investigate the impact of different thresholds on the identification of trajectory patterns and analyze the evolution of trajectory patterns in different periods.

DOI: 10.1002/smr

Table I. Change Operation Type

Level	Code Element	Operation Type
Class	Class	Add/Remove Class
	Superclass	Add/Remove/Change Superclass
	Implemented Interface	Add/Remove/Change Implemented Interface
Field	Field	Add/Remove/Type Change Field
Method	Method	Add/Remove Method
	Return Type	Return Type Change
	Parameter	Add/Remove/Type Change Parameter
Method Body	Code Fragment	Add/Remove Code Fragment
	Field Access	Add/Remove/Path Change Field Access
	Method Call	Add/Remove/Path Change Method Call

3. DEFINITION

This section defines related concepts, including change operation, change group, and trajectory pattern.

3.1. Change Operation

A change operation represents an atomic code change (e.g., adding/removing code fragment, adding/removing method parameter) as well as its evolution properties (e.g., time, developer, location). Evolution properties of change operations provide the basis for change operation grouping. Change operations can be categorized into different types as shown in Table I. These types are defined for different levels (e.g., class, method) and code elements (e.g., return type, method call). The concept of change operation is defined as follows.

Definition 1. (Change Operation) A *change operation* is a 7-tuple ($type$, $time$, $developer$, $class$, $method$, $context$, $content$), where $type$ is the type of the change operation (see Table I); $time$ is the time when the change is committed; $developer$ is the developer who commits the change; $class$ is the target class (or interface) where the change occurs; $method$ is the target method where the change occurs (not applicable for class- or field-level changes); $context$ is a set of structural dependencies of the target class and target method before the change; $content$ is the content of the change (e.g., an added/removed code fragment).

According to Table I, a change operation can be adding/removing/changing a class, field or method. For a class, the change can be adding/removing itself or adding/removing/changing its superclasses or implemented interfaces. For a field, the change can be adding/removing itself or changing its type. For a method, the change can be adding/removing itself or changing its parameters or return type. A method renaming is treated as a combination of removing the method and adding a new one. For a method body, the change can be adding/removing its code fragments or adding/removing/changing field accesses or method calls. If a method call is changed due to a parameter adding/removing change of the method, it is treated as a combination of removing the method call and adding a new one. **Path Change** of a field access or method call means the change of its control flow path in the target method. For example, moving a method call residing in a method body from its main path to a conditional branch will produce a **Path Change** of the method call. Note that a code fragment is a set of continuous statements whose length is larger than a predefined threshold. If an added or removed code fragment involves a field access or method call, an additional change operation of field access or method call is also identified.

An example of code change is presented in Figure 1. For this change, a series of change operations can be extracted as shown in Table II. For simplicity, only class/method, type, and content are listed for each change operation. In this example, the class `C1` implements a new interface `I`; the parameter `tag` of `m1` is removed; a new method call to `C2.m3` is added; a code fragment is extracted from `m1` as a separate method `m2`.

DOI: 10.1002/smr

Before	After

class C1{
...
void m1(int tag){
...
if(tag==0){
…//code fragment 1
}else{
…//code fragment 2
}
...
}
...
}
	class C1 implement I{
	...
	void m1( ){
	...
	…//code fragment 1
	C2.m3().
	...
	}
	void m2( ){
	…//code fragment 2
	}
	...
	}

Figure 1. An Example of Code Change

Table II. Change Operations Extracted from Figure 1

Class/Method	Type	Content
`C1`	Add Implemented Interface	`I`
`C1.m1`	Remove Parameter	`tag`
`C1.m1`	Remove Code Fragment	code fragment 2
`C1.m1`	Add Method Call	`C2.m3()`
`C1.m2`	Add Method	code fragment 2

3.2. Change Group

A change group is a set of change operations that are of the same type (see Table I) and at the same time share common properties. The concept of change group is defined as follows.

Definition 2. (Change Group) A *change group* is a 3-tuple $(mem, type, prop)$, where mem is a set of change operations as its members; $type$ is the common change operation type shared by all its members; $prop$ is the set of common properties shared by all its members.

Common properties of change operations are based on their evolution properties and can be considered from the following dimensions:

- *Developer*: two changes are committed by the same developer
- *Class*: the target classes of two changes are the same;
- *Superclass*: the target classes of two changes inherit the same superclass;
- *Interface*: the target classes of two changes implement the same interface;
- *Structural Dependency*: the target methods of two changes share a common method call or field access (for change operations at the method or method body level);
- *Similar Content*: the added or removed method bodies of two adding/removing method changes have similar text content, or the added or removed code fragments of two adding/removing code fragment changes have similar text content.

Note that change operations in the same group can share common properties from one or multiple dimensions. And a change operation can be included in multiple groups with different common properties.

3.3. Trajectory Pattern

Potentially relevant change groups can be further aggregated by various associations between individual change operations. The association of two change operations ch_1 and ch_2 satisfying that ch_1 is committed before or at the same time (in the same commit) of ch_2 can be represented by $ch_1 \xrightarrow{type} ch_2$, where $type$ is one of the following association types:

- *Method Call (forward)*: ch_2 is a change at the method or method body level and ch_1 is a change about the method call to ch_2's target method (Add/Remove/Path Change);

Prepared using smrauth.cls
DOI: 10.1002/smr

Table III. Associations of the Aggregation Shown in Figure 2

$group_1$	$group_2$	$type$	$insSet$
G_1	G_2	similar content	$(ch_{11}, ch_{21}), (ch_{12}, ch_{22}), (ch_{13}, ch_{23}), (ch_{14}, ch_{24})$
G_2	G_3	method call (backward)	$(ch_{21}, ch_{31}), (ch_{22}, ch_{32}), (ch_{23}, ch_{33}), (ch_{24}, ch_{33})$

- *Method Call (backward)*: ch_1 is a change at the method or method body level and ch_2 is a change about the method call to ch_1's target method (Add/Remove/Path Change);
- *Field Access (forward)*: ch_2 is a change about a field and ch_1 is a change about the access to the field (Add/Remove/Path Change);
- *Field Access (backward)*: ch_1 is a change about a field and ch_2 is a change about the access to the field (Add/Remove/Path Change);
- *Inheritance/Interface Implementation (forward)*: ch_1 and ch_2 are both class-level changes and ch_1's target class inherits (implements) ch_2's target class (interface) before or after the change;
- *Inheritance/Interface Implementation (backward)*: ch_1 and ch_2 are both class-level changes and ch_2's target class inherits (implements) ch_1's target class (interface) before or after the change;
- *Same Dependency*: ch_1 and ch_2 are both changes about method call/field access but of different change operation types (e.g., one is adding method call and the other is removing method call) and the called method/accessed field are the same;
- *Similar Content*: ch_1 and ch_2 are both changes about adding/removing methods or code fragments but of different change operation types (e.g., one is adding method and the other is removing method) and the methods or code fragments are similar (a special case is that ch_1 and ch_2 are changes about the same methods or code fragments).

Note that if ch_1 and ch_2 are in the same commit, we assume that they occur in a certain order. For *Method Call*, we assume that the change of method or method body occurs before the change of method call. For *Field Access*, we assume that the change of field occurs before the change of field access. For *Inheritance/Interface Implementation*, we assume that the change of the base class (interface) occurs before the change of the derived class. In this way, we can assure that two change operations in the same commit do not produce both a forward and a backward associations of the same type.

In general, two change groups with a number of association instances of the same type can be associated as the following definition of change group association.

Definition 3. (Change Group Association) A *change group association* is a 4-tuple $(group_1, group_2, type, insSet)$, where $group_1$ and $group_2$ are two different change groups; $type$ is an association type; $insSet$ is an association instance set, i.e., a set of change operation pairs (ch_1, ch_2) satisfying $ch_1 \in group_1 \wedge ch_2 \in group_2 \wedge ch_1 \xrightarrow{type} ch_2$.

A trajectory pattern consists of a set of change groups that are aggregated by various associations between them. The concept of trajectory pattern is defined as follows.

Definition 4. (Trajectory Pattern) A *trajectory pattern* is a 2-tuple $(groups, associations)$, where $groups$ is a set of change groups; $associations$ is a set of change group associations and for each $asso \in associations$ there is $asso.group_1, asso.group_2 \in groups$.

The concept of trajectory pattern can be illustrated with the example shown in Figure 2. In this example, some of the change operations in the change group G_1 have similar change content with those in G_2. If the number of *Similar Content* associations existing between G_1 and G_2 exceeds a given threshold, these two groups can be aggregated together. Similarly, G_2 and G_3 are aggregated together due to the associations of *Method Call (backward)*. This trajectory pattern can be represented by a 2-tuple $(groups, associations)$, where $groups = \{G_1, G_2, G_3\}$ and $associations$ is the set of associations shown in Table III. And the whole trajectory pattern can be interpreted as: Jack extracts some code fragments of the class `Payment1` as separate common methods in the class `Util`, then Tom changes some methods in the class `Payment2` to add method calls to the extracted methods.

DOI: 10.1002/smr

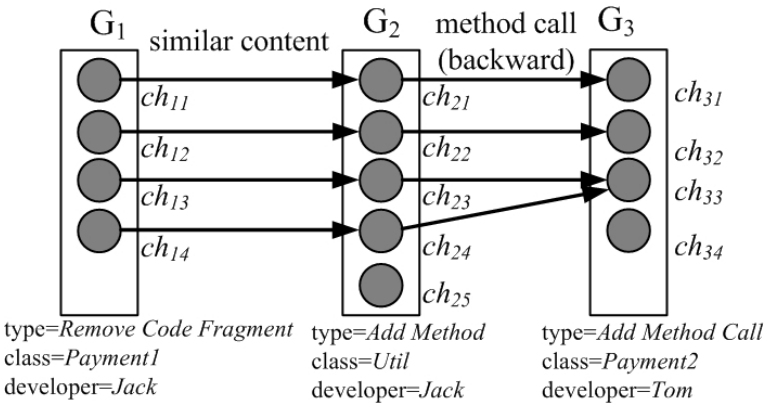

Figure 2. An Example of Change Group Aggregation

4. APPROACH

SETGA takes as input a series of commit records from version control systems and produces a set of trajectory patterns. This section introduces the three steps of SETGA, i.e., change operation extraction, grouping, and aggregation.

4.1. Change Operation Extraction

Different from the fine-grained (statement and variable level) source code changes extracted by the change distillling algorithm [1], SETGA extracts change operations at various levels such as class, method, and code fragment.

For each commit, SETGA analyzes all the involved files and generates one or multiple change operations for each file by comparing its source code before and after the change.

To extract change operations, SETGA first determines whether some new files are added or some old files are removed according to the commit records. If found, change operations with the type of adding/removing classes will be generated. Then SETGA analyzes the ASTs of each involved file before and after the change. This analysis maps between class/field/method nodes in the two ASTs by calculating the similarity of their names and signatures. For two mapped method nodes, SETGA further maps between their parameters by comparing their names and types. Based on the mapping, SETGA extracts class-, field-, and method-level change operations.

After that, SETGA analyzes the method body of each mapped method to extract method-body-level change operations. It compares the field accesses and method calls of a method in the two ASTs and identifies added/removed field accesses and method calls. For a field access or method call that exists in both the two ASTs, SETGA further compares its control flow paths in the two ASTs and determines whether there is a change of path. To identify added and removed code fragments, SETGA uses text-based diff algorithms to compare the source code of a method before and after the change. For each added or removed code fragment, if its length reaches a predefined threshold $threshold_{frag}$, a change operation is generated with its source code as the change content.

4.2. Change Operation Grouping

Change operation grouping is based on the identification of common properties among different change operations. For structural dependency, SETGA does not consider dependencies to third-party libraries (e.g., JDK) as common properties. And to determine the content similarity of change operations, SETGA uses semantic clustering [12], which groups source artifacts that use similar vocabulary, to cluster all the added/removed method bodies and code fragments in extracted change operations. Those method bodies and code fragments are transformed into a corpus of documents. Then SETGA uses the bisecting K-means clustering algorithm implemented in our

DOI: 10.1002/smr

previous work [15] to generate a set of clusters. All the method bodies and code fragments in the same cluster are regarded to be similar.

SETGA uses a change operation grouping algorithm (see Algorithm 1) to group similar change operations together. The algorithm takes as input a set of change operations ($ChOpSet$). It returns a set of change groups ($Groups$), each of which consists of change operations of the same type and sharing some common properties. For each change operation $change$ in $ChOpSet$, the function $prop(change)$ returns a set of properties of $change$ that can be considered when computing the common properties with other change operations (see Section 3.2).

Three predefined thresholds are used in the algorithm: $threshold_{prop}$ specifies the minimum number of common properties of a group; $threshold_{minIns}$ specifies the minimum number of instances (change operations) in a group; $threshold_{maxIns}$ specifies the maximum number of instances (change operations) in a group. The reason for setting a threshold for the maximum number of instances is that a change group with too many instances usually represents trivial commonalities such as methods modified by the same developer.

The algorithm first initializes $Groups$ to an empty set (Line 2), and then executes an iterative process to group change operations (Line 3-26). For each change operation $change$, the algorithm tries to merge it with existing groups in $Groups$ (Line 4-19). For each group G with the same type of $change$, if $change$ possesses all the common properties of G, $change$ is added to G (Line 6-7). Otherwise, if the number of their common properties reaches the threshold $threshold_{prop}$, a new group is created with G's members and $change$ as its members, their common type as its type, their common property set as its property set (Line 9-13). After all the groups in $Groups$ are considered, a new group is created with $change$ as its only member (Line 17-18). And a procedure $filterOutSubgroup$ is invoked to filter out the groups that are subgroups of some other groups (Line 19). Note that we say a group G_1 is the subgroup of another group G_2 if the following condition holds: $G_1.mem \subseteq G_2.mem \wedge G_1.prop \subseteq G_2.prop \wedge G_1.type = G_2.type$. After all the change operations in $ChOpSet$ are considered, all the produced groups in $Groups$ are checked and the groups whose numbers of members are smaller than $threshold_{minIns}$ or larger than $threshold_{maxIns}$ are eliminated (Line 21-25).

Algorithm 1 Change Operation Grouping Algorithm

```
1:  function GROUPCHANGEOPERATION(ChOpSet)
2:      Groups = { }
3:      for each change ∈ ChOpSet do
4:          for each G ∈ Groups do
5:              if change.type == G.type then
6:                  if G.prop ⊆ prop(change) then
7:                      G.mem = G.mem ∪ {change}
8:                  else
9:                      Common = prop(change) ∩ G.prop
10:                     if |Common| ≥ threshold_prop then
11:                         NG = newGroup(G, change)
12:                         Groups = Groups ∪ {NG}
13:                     end if
14:                 end if
15:             end if
16:         end for
17:         SingleChG = newGroup(change)
18:         Groups = Groups ∪ {SingleChG}
19:         filterOutSubgroup(Groups)
20:     end for
21:     for each G ∈ Groups do
22:         if |G.mem| < threshold_minIns ∨ |G.mem| > threshold_maxIns then
23:             Groups = Groups − {G}
24:         end if
25:     end for
26:     return Groups
27: end function
```

Figure 3 shows an example of the process of change operation grouping with the following threshold settings: $threshold_{prop} = 2$, $threshold_{minIns} = 2$, $threshold_{maxIns} = 3$. The input

DOI: 10.1002/smr

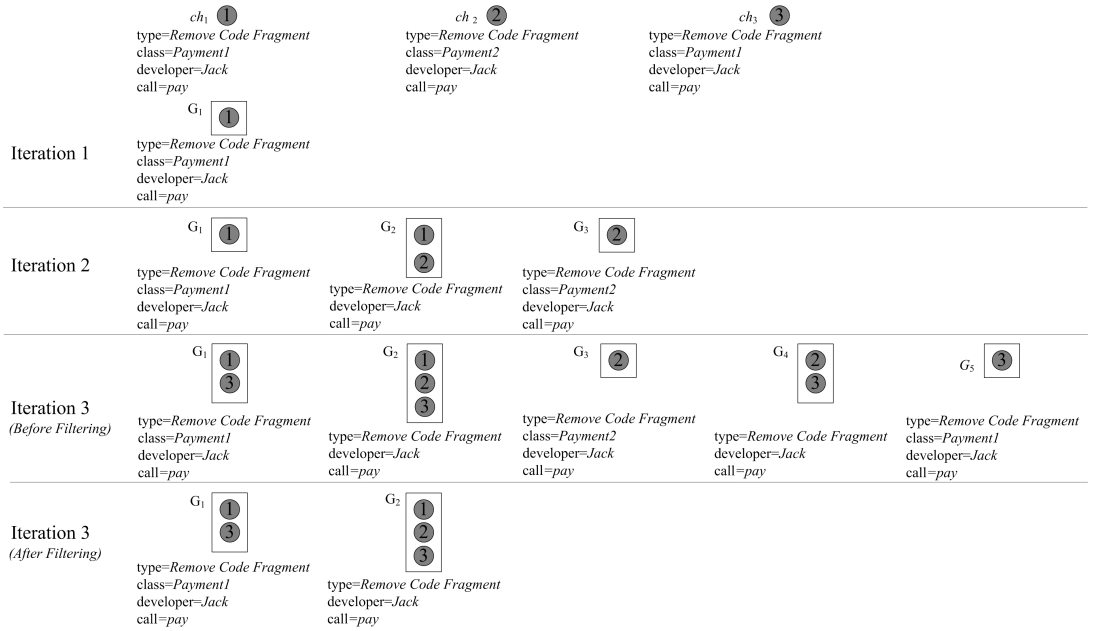

Figure 3. An Example of Change Operation Grouping

of the process is a set of three change operations (ch_1, ch_2, and ch_3) of the type *Remove Code Fragment*. These three change operations are committed by the same developer and their target methods all have a dependency of calling method `pay`. Moreover, ch_1 and ch_3 have the same target class. The grouping process of these change operations is conducted in an iterative way. In the first iteration, a new group (G_1) with ch_1 as its only member is created. The common properties of the group are exactly the same with the properties of ch_1. In the second iteration, a new group (G_2) is created by merging G_1 and ch_2, which has ch_1 and ch_2 as its members and their common properties (*type*, *developer*, and *call pay()*) as its properties. Besides, a new group (G_3) with ch_2 as its only member is also created. In the third iteration, ch_3 is added to G_1, since it possesses all the common properties of G_1. Similarity, ch_3 is also added to G_2. As ch_3 does not possess all the common properties of G_3 but shares three common properties with it, a new group (G_4) is created by merging G_3 and ch_3. Besides, a new group (G_5) with ch_3 as its only member is also created. During the filtering step of this iteration, G_4 is removed as it is the subgroup of G_2; G_5 is removed as it is the subgroup of G_1. Finally, G_3 is removed since its number of members is only one (smaller than $threshold_{minIns}$) and we get two change groups as the results of change operation grouping.

4.3. Change Group Aggregation

SETGA uses a change group aggregation algorithm (See Algorithm 4) to identify trajectory patterns. The algorithm is defined based on two functions, i.e., change group association (See Algorithm 2) and aggregation extension (See Algorithm 3). Before introducing the change group aggregation algorithm, we will first describe the change group association function and the aggregation extension function.

The change group association function described in Algorithm 2 returns a set of change group associations from a change group G to another change group G'. As there may be multiple types of associations between G and G', the returned set may include multiple change group associations of different types. In the algorithm, $AssoTypes$ is a set consisting of all the association types such as *Method Call (forward)*, *Method Call (backward)*, and *Similar Content* (see Section 3.3). For each association type $type$, a change group association $asso$ is generated (Line 3-13). Its elements $asso.group_1$, $asso.group_2$ and $asso.type$ are set to G, G', and $type$, respectively (Line 4). And $asso.insSet$ is set to an empty set. For each $ch \in G.mem$ and each $ch' \in G'.mem$, the function $ExistAssociation(type, ch, ch')$ checks whether there exists an association relationship of the type

DOI: 10.1002/smr

$type$ from ch to ch' (Line 7). If exists, the pair (ch, ch') is added into $asso$'s instance set (i.e., $asso.insSet$) (Line 8).

Algorithm 2 Change Group Association Algorithm

```
1: function ASSOCIATEGROUPS(G, G')
2:     AssoSet = { }
3:     for each type ∈ AssoTypes do
4:         asso = newAssociation(G, G', type)
5:         for each ch ∈ G.mem do
6:             for each ch' ∈ G'.mem do
7:                 if ExistAssociation(type, ch, ch') then
8:                     asso.insSet = asso.insSet ∪ {(ch, ch')}
9:                 end if
10:            end for
11:        end for
12:        AssoSet = AssoSet ∪ {asso}
13:    end for
14:    return AssoSet
15: end function
```

The aggregation extension function described in Algorithm 3 takes as input a trajectory pattern agg and a change group association $asso$, and returns a new trajectory pattern $newAgg$ that extends agg with $asso$. The extension adds $asso$ and a new change group involved in $asso$ into agg. Correspondingly, association instances that do not conform to the new trajectory pattern are filtered out from $newAgg$. If the extension does not exist, it returns $Null$.

The algorithm first checks whether the basic assumption of aggregation extension is satisfied, i.e., one change group involved in $asso$ is included in $agg.groups$ while the other is not (Line 2-8). If true, the change group that is included in $agg.groups$ and the other one that is not are represented by G_1 and G_2 respectively. Otherwise, the algorithm returns $Null$. Then a new trajectory pattern $newAgg$ is created with $agg.groups$ and G_2 as its change groups, a copy of $agg.associations$ and $asso$ as its association set (Line 9-12).

After that, an iterative process is conducted to filter out association instances that do not conform to the new trajectory pattern (i.e., $newAgg$) (Line 13-39). For each change group G in $newAgg$, the algorithm filters its association instances with other groups in the following way. It first computes an association set $assoSet$, which is the subset of $newAgg.associations$ that involves G (Line 17-22). It then filters in turn each association in $assoSet$ (Line 23-38). The association instances who have at least one end that is not in any assocation in $assoSet$ should be removed. For each association as in $assoSet$ between G and another change group G', the function $removeIns(as, G, G', assoSet)$ is used to compute the association instances that need to be removed from $as.insSet$ (Line 29). The function returns a set of change operation pairs $(ch_1, ch_2) \in as.insSet$ that satisfy either of the following two conditions:

- $ch_1 \in G.mem \wedge ch_2 \in G'.mem \wedge (\exists a \in assoSet, (\nexists ch, (ch_1, ch) \in a.insSet \vee (ch, ch_1) \in a.insSet))$
- $ch_1 \in G'.mem \wedge ch_2 \in G.mem \wedge (\exists a \in assoSet, (\nexists ch, (ch_2, ch) \in a.insSet \vee (ch, ch_2) \in a.insSet))$

The returned change operation pairs are removed from $as.insSet$. Finally, if the size of $as.insSet$ after filtering is smaller than $threshold_{minIns}$, which means the new trajectory pattern $newAgg$ has not enough instances, the algorithm returns $Null$. After all the associations of $newAgg$ have been filtered, it is returned as the extended trajectory pattern.

Figure 4 shows an example of the aggregation extension process with the following threshold settings: $threshold_{minIns} = 2$. The inputs of the process are a trajectory pattern agg and a change group association as_3. agg includes three groups G_1, G_2 and G_3, which are associated by two change group associations as_1 and as_2. as_3 is a change group association between G_3 and G_4. In the extension process, a new trajectory pattern $newAgg$ is created with G_1, G_2, G_3 and G_4 as its change groups and as_1, as_2 and as_3 as its change group associations. Then an iterative process is conducted to filter out association instances in $newAgg$ that do not conform to $newAgg$. In the

DOI: 10.1002/smr

Algorithm 3 Aggregation Extension Algorithm

```
1: function EXTENDAGGREGATION(agg, asso)
2:     if asso.group_1 ∈ agg.groups ∧ asso.group_2 ∉ agg.groups then
3:         G_1 = asso.group_1, G_2 = asso.group_2
4:     else if asso.group_1 ∉ agg.groups ∧ asso.group_2 ∈ agg.groups then
5:         G_1 = asso.group_2, G_2 = asso.group_1
6:     else
7:         return Null
8:     end if
9:     newAgg = newAggregation()
10:    newAgg.groups = agg.groups ∪ {G_2}
11:    assoSet = agg.associations ∪ {asso}
12:    newAgg.associations = copyAssos(assoSet)
13:    queue = [ ], doneGroups = {G_1, G_2}
14:    queue.enQueue(G_1)
15:    while queue.length > 0 do
16:        G = queue.deQueue()
17:        assoSet = { }
18:        for each as ∈ newAgg.associations do
19:            if as.group_1 == G ∨ as.group_2 == G then
20:                assoSet = assoSet ∪ {as}
21:            end if
22:        end for
23:        for each as ∈ assoSet do
24:            if as.group_1 == G then
25:                G' = as.group_2
26:            else
27:                G' = as.group_1
28:            end if
29:            temp = removeIns(as, G, G', assoSet)
30:            as.insSet = as.insSet − temp
31:            if |as.insSet| < threshold_minIns then
32:                return Null
33:            end if
34:            if G' ∉ doneGroups then
35:                queue.enQueue(G')
36:                doneGroups = doneGroups ∪ {G'}
37:            end if
38:        end for
39:    end while
40:    return newAgg
41: end function
```

first iteration, G_3 and its related associations as_2 and as_3 are considered. For as_2, the association instance between ch_{23} and ch_{33} and the one between ch_{24} and ch_{34} are removed from the instance set of as_2, as they have no corresponding association instances in the instance set of as_3. In the second iteration, G_2 and its related associations as_1 and as_2 are considered. For as_1, the association instance between ch_{11} and ch_{23} and the one between ch_{12} and ch_{24} are removed from the instance set of as_1, as they have no corresponding association instances in the instance set of as_2. Finally, a new trajectory pattern ($newAgg$ after filtering, as shown in Figure 4), which extends agg with as_3, is returned as the result.

Based on Algorithm 2 and Algorithm 3, Algorithm 4 identifies trajectory patterns by aggregating change groups. It takes as input a set of change groups $Groups$ and returns a set of trajectory patterns (i.e., aggregations of change groups) $AggSet$. The algorithm first generates all the binary associations between change groups in $Groups$ (Line 3-14). For each $G, G' \in Groups$, the algorithm identifies possible change group associations from G to G' using the $associateGroups(G, G')$ function (See Algorithm 2) (Line 5). For each returned association, if the size of its instance set (i.e., $insSet$) is not smaller than $threshold_{minIns}$, it is added to the binary association set $BinAssos$ and a candidate trajectory pattern with two groups (G and G') is created and put into a queue (Line 6-12).

Next, an incremental and iterative aggregation process is conducted until the queue is empty (Line 15-30). In each iteration, the algorithm dequeues a candidate pattern agg and tries to extend it

DOI: 10.1002/smr

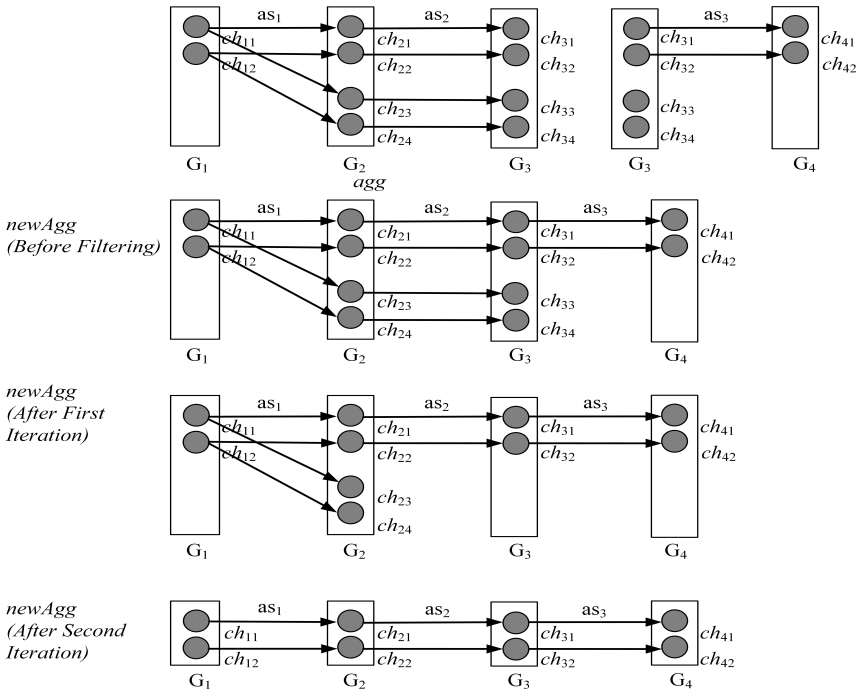

Figure 4. An Example of Aggregation Extension

with each binary association $asso$ in $BinAssos$ (Line 18-26). The $extendAggregation(agg, asso)$ function (See Algorithm 3) is used to generate a new trajectory pattern that extends agg with $asso$. If a new trajectory pattern agg' is generated, it is put into the queue (Line 21). If there's no association instance removed from agg during the process of $extendAggregation$, agg can be replaced with agg'. In this case, the algorithm further checks whether agg is contained in agg' using the function $contain(agg', agg)$ (Line 22), which returns true if all the association instances of agg is contained in agg'. If agg is not contained in any trajectory pattern that is extended from agg, it is added into the returned pattern set (Line 27-29).

5. EMPIRICAL STUDY

To evaluate whether SETGA can effectively summarize evolution trajectory from dispersed code changes, we conducted an empirical study with three open-source systems to investigate the following five research questions:

RQ1 What kinds of trajectory patterns can be identified from these systems? What associations do they reflect?

RQ2 How do the identified trajectory patterns span code changes at different times, in different locations, and by different developers?

RQ3 How do the thresholds impact the identification of different kinds of trajectory patterns?

RQ4 What underlying rules and problems can be revealed by the identified trajectory patterns?

RQ5 How do the trajectory patterns evolve over time? What additional rules and problems can be further revealed by the evolution analysis of trajectory patterns?

DOI: 10.1002/smr

Algorithm 4 Change Group Aggregation Algorithm

```
1: function AGGREGATECHANGEGROUP(Groups)
2:     AggSet = { }, BinAssos = { }, queue = [ ]
3:     for each G ∈ Groups do
4:         for each G′ ∈ Groups do
5:             AssoSet = associateGroups(G, G′)
6:             for each asso ∈ AssoSet do
7:                 if |asso.insSet| ≥ threshold_minIns then
8:                     agg = newAggregation(G, G′, asso)
9:                     queue.enQueue(agg)
10:                    BinAssos = BinAssos ∪ {asso}
11:                end if
12:            end for
13:        end for
14:    end for
15:    while queue.length > 0 do
16:        agg = queue.deQueue()
17:        contained = False
18:        for each asso ∈ BinAssos do
19:            agg′ = extendAggregation(agg, asso)
20:            if agg′ ≠ Null then
21:                queue.enQueue(agg′)
22:                if contain(agg′, agg) == True then
23:                    contained = True
24:                end if
25:            end if
26:        end for
27:        if contained == False then
28:            AggSet = AggSet ∪ {agg}
29:        end if
30:    end while
31:    return AggSet
32: end function
```

5.1. Basic Results

In our empirical study, we applied SETGA to three open-source systems, i.e., jEdit[‡], jBPM[§], Eclipse SWT[¶]. These systems are popularly used for evolution analysis in literatures. jEdit is a small text editor with about 80 thousand lines of code; jBPM is a medium-sized business process management suite with about 130 thousand lines of code; Eclipse SWT is a large widget toolkit for Java with about 500 thousand lines of code. This selection of subject systems covers small-, medium-, and large-sized systems. All the three projects have Git repositories available and have successive releases in their master branches, which facilitates the analysis of the evolution of trajectory patterns in different periods. For these three systems, we analyzed their commit records from the master branches of Git repositories obtained from SourceForge, GitHub, and Eclipse.org, respectively.

The identification of trajectory patterns is influenced by the following thresholds: $threshold_{frag}$ (minimum length of a code fragment by character), $threshold_{prop}$ (minimum number of common properties of a group), $threshold_{minIns}$ (minimum number of instances in a group), $threshold_{maxIns}$ (maximum number of instances in a group). After analyzing the data, we found that if $threshold_{frag}$ is too small (e.g., lower than 100 characters), the clustering algorithm will produce too many small and meaningless similar code fragments, thus introduce too much noise for the identification of trajectory patterns. As change operations sharing only one common property (e.g., committed by the same developer) often involve no meaningful commonality, we require that a change group should share at least two common properties (i.e., $threshold_{prop} = 2$). For $threshold_{minIns}$, we take the lowest requirement, i.e., requiring that a change group has at least two instances. For $threshold_{maxIns}$, our data analysis showed that when it is set to 11 or larger, a

[‡] jEdit: http://jedit.org/
[§] jBPM: http://jbpm.jboss.org/
[¶] Eclipse SWT: http://www.eclipse.org/swt/

DOI: 10.1002/smr

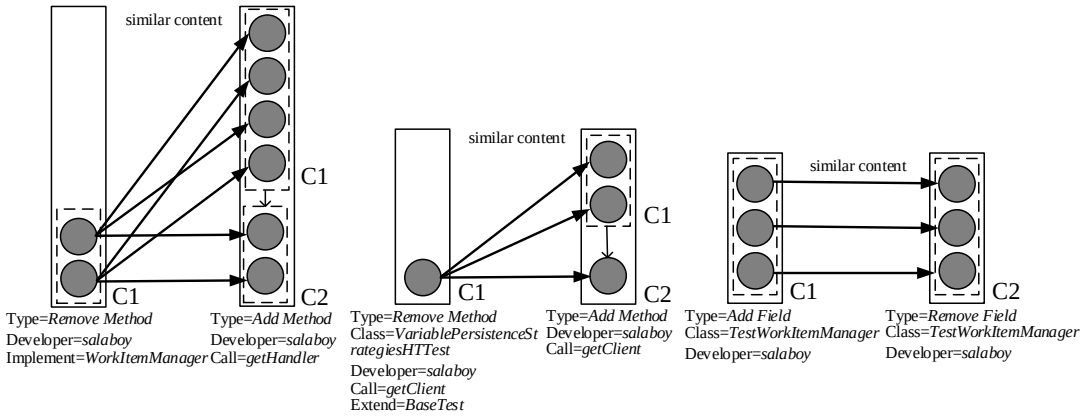

Figure 5. An Example of Commits to Groups

large part of the identified change groups are meaningless ones with only two common properties (e.g., the same class and the same developer). Based on the above consideration, we used the following thresholds in the study: $threshold_{frag} = 100$, $threshold_{prop} = 2$, $threshold_{minIns} = 2$, $threshold_{maxIns} = 10$.

The basic results of the study are shown in Table IV. For each period, the table lists the number of commits (#C), average number of files involved in each commit (#F/C), number of extracted change operations (#O), number of groups (#G), average number of common properties in each group (#P/G), average number of change operations in each group (#O/G), number of trajectory patterns (#T), average number of change groups in each pattern (#G/T), average number of change operations in each pattern (#O/T), average number of commits involved in each pattern (#C/T), average number of patterns involved in each commit (#T/C), execution time by minute (#M).

From the table, it can be seen that most of the evolution periods lasted for several months to one year except jEdit 4.1-4.2 (1.5 years). This makes that jEdit 4.1-4.2 has much more development tasks than the others and thus produces much more commits, change operations, groups, and trajectory patterns. The number of commits ranges from 31 to 1,545, the average number of files involved in each commit ranges from 1.25 to 6.97, the number of extracted change operations ranges from 1,175 to 9,900. For each period, 65 to 577 trajectory patterns were identified, and each pattern involves 2.30-9.72 change groups, 5.92-24.40 change operations, 1.67-17.86 commits on average. On the other hand, each commit is involved in 3.00-12.89 patterns on average.

It can be seen that it is common that a pattern involves change operations from multiple commits and the change operations of a commit are involved in multiple patterns. An example from jBPM is shown in Figure 5. It includes three trajectory patterns which involve two commits (C1 and C2). According to the commit logs, the developers revised the implementation of the functionality of "variable persistence strategies for Human Task in Async Handlers and corresponding test code in these two commits. It can be seen that C2 continued the functionality implementation and testing of C1, which is reflected by the fact that change operations from C1 and C2 are involved in the same trajectory patterns. On the other hand, the change operations from C1 are included in three trajectory patterns, each of which reflects the testing of a different aspect of the implemented functionality.

After analyzing the data, we found that usually more than 50% of the extracted change operations are involved in change groups; 10%-20% of the extracted change operations are involved in trajectory patterns (i.e., systematic changes). The numbers of identified patterns highly depend on the numbers of commits and extracted change operations.

The time performance of SETGA for different periods of the three systems is shown in Table IV (the last column) in terms of execution time. The experiments were conducted on a ThinkPad X230 laptop with two Intel Core i5-3230M 2.60 GHz processors and 8 GB RAM, running Windows 8.1. It can be seen that in most cases SETGA can mine all the trajectory patterns in one hour. And the execution time mainly depends on the numbers of extracted change operations.

DOI: 10.1002/smr

Table IV. Basic Results of the Empirical Study

System	Period	Time	#C	#F/C	#O	#G	#P/G	#O/G	#T	#G/T	#O/T	#C/T	#T/C	#M
jEdit	3.0-3.1	Dec. 2000 - Apr. 2001	31	6.97	1175	501	4.41	4.39	65	3.06	7.31	2.13	3.00	2
	3.1-3.2	Apr. 2001 - Aug. 2001	110	4.98	3167	976	4.54	4.30	132	2.67	6.49	2.24	7.90	5
	3.2-4.0	Aug. 2001 - Apr. 2002	341	3.68	7136	2167	4.22	4.43	377	4.73	12.55	4.12	11.22	52
	4.0-4.1	Apr. 2002 - Feb. 2003	369	3.31	5648	1511	3.71	4.03	231	2.56	6.26	2.63	9.22	17
	4.1-4.2	Feb. 2003 - Aug. 2004	575	3.56	9900	2555	3.84	4.09	569	7.59	20.73	17.86	12.89	147
jBPM	5.1-5.2	Jun. 2011 - Dec. 2011	324	1.66	3328	858	3.68	4.12	71	2.30	5.92	2.10	8.33	3
	5.2-5.3	Dec. 2011 - May 2012	322	2.16	6961	1258	3.53	3.99	135	4.00	11.14	2.63	4.18	11
	5.3-5.4	May 2012 - Nov. 2012	288	2.46	4608	996	4.62	4.17	95	2.62	6.69	1.67	5.96	7
SWT	3.5-3.6	Jun. 2009 - Jun. 2010	1545	1.25	9077	2626	3.68	4.12	577	9.72	24.40	4.42	9.82	78
	3.6-3.7	Jun. 2010 - Jun. 2011	777	1.71	6113	1641	3.71	3.74	190	2.80	6.75	2.62	8.34	22

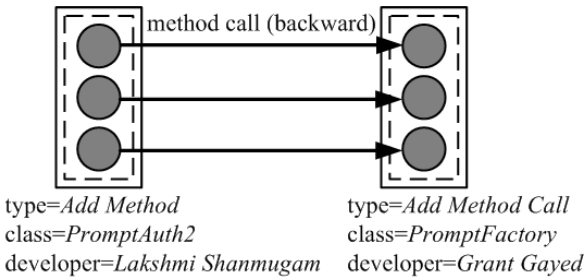

Figure 6. An Example of *Add/Remove Method/Field* (Type 1)

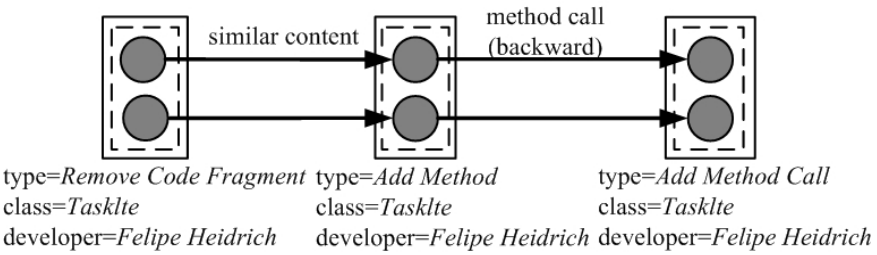

Figure 7. An Example of *Extract Method* (Type 2)

5.2. RQ1: Categories of Identified Trajectory Patterns

After analyzing all the 2,442 trajectory patterns identified from the three systems, we found that most of them can be categorized into the following six basic types based on the involved association types . In the following examples, arrows within a change group (rectangle) represent time orders between change operations; a dashed box represents that all the change operations in it occur in the same commit. And to save space, only a part of common properties are listed for some change groups.

1) *Add/Remove Method/Field (Type 1: A/R Method/Field)*: A method/field is added to/removed from a class; another method adds/removes a method call/field access to the method/field. An example from Eclipse SWT is shown in Figure 6. In this example, a developer created three methods in the `PromptAuth2` class to support COM interfaces; another developer added method calls to these new methods in three methods of the class `PromptFactory`.

2) *Extract Method (Type 2: E Method)*: A code fragment f is extracted from an existing method m_1 and used to create a new method m_2; some other methods add method calls to m_2. A variant of this pattern is that some other methods add method calls to m_1 instead of m_2, which means the common implementation required by other methods remains in m_1. An example from Eclipse SWT is shown in Figure 7. In this example, a developer extracted two code fragments from two methods of the `Tasklte` class as separate methods, and added method calls from the original methods to the two new methods.

3) *Change Method Signature (Type 3: C Signature)*: A method m_1's signature is changed as well as its method body; another method m_2 adds a new method call, or removes its method call, or changes its path of method call to m_1. An example from jEdit is shown in Figure 8. In this example,

DOI: 10.1002/smr

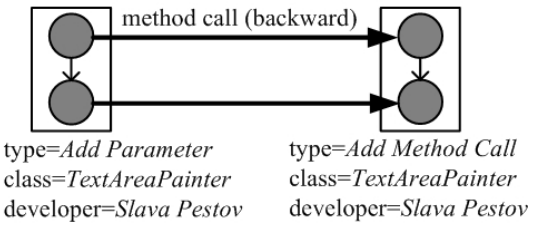

Figure 8. An Example of *Change Method Signature* (Type 3)

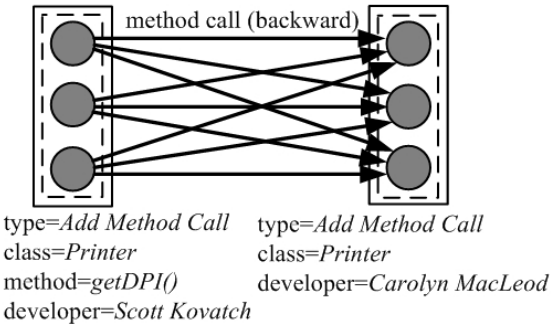

Figure 9. An Example of *Change Method Contract* (Type 4)

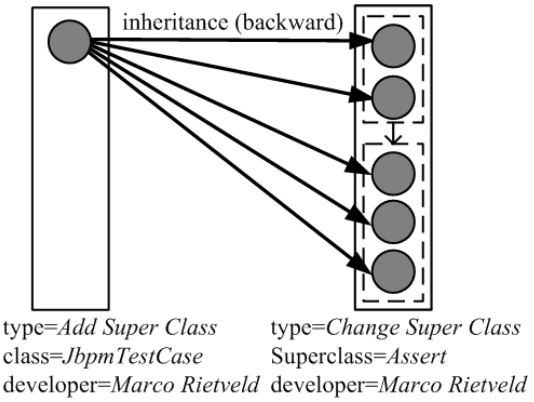

Figure 10. An Example of *Change Inheritance Hierarchy* (Type 5)

a developer added a parameter to two methods in the `TextAreaPainter` class and then modified two method calls to these two methods.

4) *Change Method Contract (Type 4: C Contract)*: A method m_1's method body is changed; another method m_2 adds a new method call, or removes its method call, or changes its path of method call to m_1. The change to m_1's method body in this case usually implies a change to its contract. An example from Eclipse SWT is shown in Figure 9. In this example, a developer revised the functionalities of the `getDPI` method; another developer revised another three methods to add calls to the `getDPI` method.

5) *Change Inheritance Hierarchy (Type 5: C Inheritance)*: Some class-, field-, or method-level changes occur in a class C_1; another class C_2 adds an inheritance, removes its inheritance, or change its inheritance to C_1. In some cases, some additional changes occur in C_2 after the change of its inheritance. An example from jBPM is shown in Figure 10. In this example, a developer added an inheritance from the class `JbpmTestCase` to `Assert`, and then changed the superclasses of `Assert`'s five subclasses to `JbpmTestCase` in two commits.

6) *Repeated Adding/Removing Methods/Method Calls (Type 6: R A/R Method/Call)*: A method or method call is first added/removed and then removed/added. An example from jBPM is shown in

DOI: 10.1002/smr

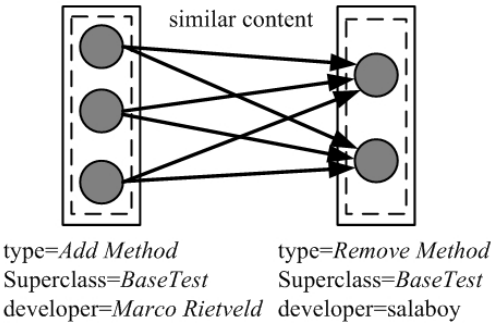

Figure 11. An Example of *Repeated Adding/Removing Methods/Method Calls* (Type 6)

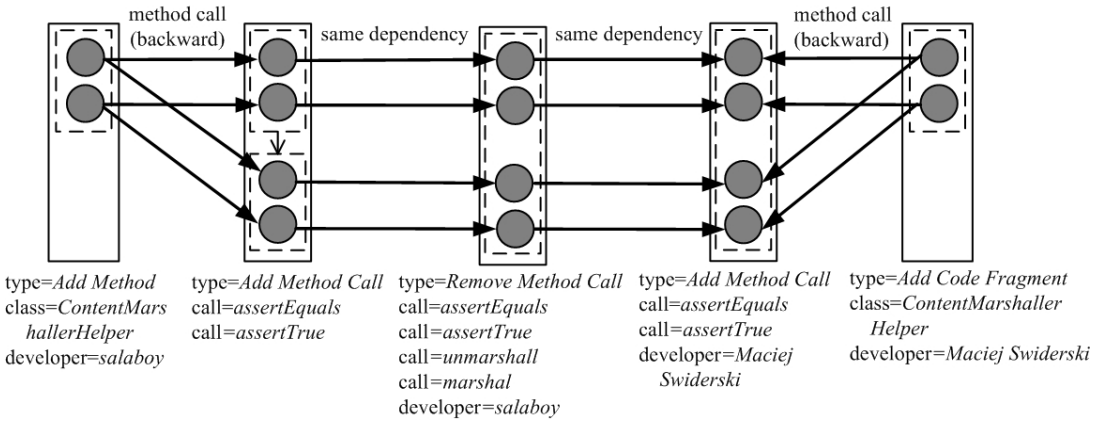

Figure 12. An Example of Combination of Different Trajectory Pattern Types

Figure 11. In this example, a developer added three similar methods to the subclasses of `BaseTest`; another developer removed two of them.

Note that the above six types of trajectory patterns are basic types. A specific trajectory pattern can be a combination of different types. For example, Figure 12 shows a combined trajectory pattern from jBPM, which reflects the following process: a developer *salaboy* added two methods (`marshal` and `ummarshall`) to the class `ContentMarshallerHelper`; to test these two methods, *Maciej Swiderski* and *salaboy* added method calls to them in test classes; as the tests failed, *salaboy* removed method calls to `marshal` and `ummarshall` from test classes; to fix the problem, *Maciej Swiderski* added code fragments to `marshal` and `ummarshall` and added method calls to them in test classes. In summary, *Maciej Swiderski* fixed some bugs in two methods introduced by *salaboy* and confirmed the revision by unit testing.

The distributions of different types of trajectory patterns in the three subject systems are shown in Figure 13. Based on the above definition, the types of a trajectory pattern can be automatically determined. Note that the sum of the percentages of different pattern types of a subject system is higher than 100%, as a trajectory pattern can be categorized into multiple basic types.

It can be seen that only a small part of the patterns (about 10%) cannot be categorized into any basic types. These patterns are usually atypical combinations of specific changes, for example the paths of a group of method calls are first changed and then removed. Only a few of patterns (about 1%) involve extracting methods (Type 2). A large part of patterns can be categorized into Type 1 and Type 4, which represent ordinary changes related to adding/removing/changing methods or fields. There are also a large part of patterns involving repeated adding/removing the same methods or methods calls (Type 6), which reflect various causes such as recovering accidentally deleted code and refactoring newly added methods. A large part (about 20%) of trajectory patterns identified from jEdit and Eclipse SWT are related to changing method signature (Type 3), while only a small part (about 3%) of those identified from jBPM are related to this type. A small part (lower than

Prepared using smrauth.cls
DOI: 10.1002/smr

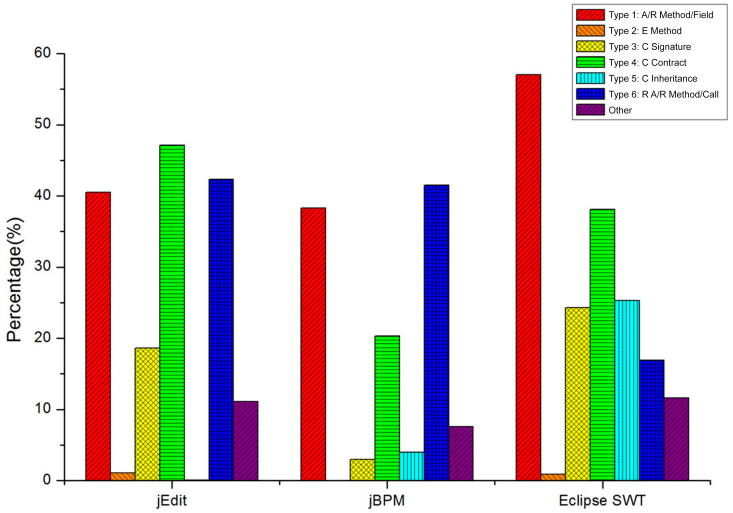

Figure 13. Distribution of Different Types of Trajectory Patterns

5%) of patterns identified from jEdit and jBPM are related to changing inheritance hierarchy (Type 5), while a large part (about 25%) of those identified from Eclipse SWT are related to this type. After analyzing the commit logs, we found Eclipse SWT did a lot of inheritance-related refactoring during the periods.

The trajectory patterns identified from jBPM differ from those identified from the other two projects that a large part of the patterns are related to testing. jBPM employed test-driven development during the analyzed periods. Therefore, many trajectory patterns involve testing-related changes such as defining new methods and related test cases and then implementing the methods.

5.3. RQ2: Span of Identified Trajectory Patterns

To answer this question, we analyzed the span of the identified trajectory patterns from the aspects of time, location, and developer. The span of time measures the days between the earliest and the latest code changes involved in a pattern. The span of location measures the number of files that are revised in the code changes involved in a trajectory pattern. The span of developer measures the number of developers who commit at least one code change involved in a trajectory pattern. The results of our dispersion analysis are shown in Table V. For each measure, the table lists the maximum, median, and average value. The minimum values are not listed as they are the same for all the systems and periods: 0 for #Day (the same commit); 1 for #File (the same file); 1 for #Developer (the same developer).

It can be seen that most of the identified patterns span several to dozens of days. For some periods (e.g., jEdit 4.1-4.2), most of the patterns span over several months. And it can be seen that there is a correlation between the time span of trajectory patterns and the length of release cycles. Usually the longer the development cycle, the longer the identified trajectory patterns span.

Most of the identified patterns span only one file except jBPM 5.2-5.3 and Eclipse SWT 3.5-3.6, where most of the patterns span two and three files respectively. And overall, among all the identified patterns, 36.4% involve two files or more, 16.7% involve three files or more, and 5.7% involve five files or more.

Most of the identified patterns involve only one developer. For jEdit, the whole project involved only one developer from version 3.0-3.2 and three developers from version 3.2-4.2. For jBPM 5.1-5.4, about 23.4% of the identified patterns involve two developers and 7.6% of them involve three or more. For Eclipse SWT 3.5-3.7, about 20.4% of the patterns involve two developers and 20.5% of them involve three or more.

DOI: 10.1002/smr

The analysis of time span shows that code changes involved in a trajectory pattern can span over several months. This often reflects an adjustment of requirements or technical decisions in a long time after initial code changes. For example, during jEdit 4.1-4.2, to add a new tool bar a developer added two methods in the `ToolBarOptionPane` class; four months later the developer merged the two methods to another two methods respectively to improve the design of the tool bar option panel. In some cases, this may also reflect a systematic change conducted off and on in a long time. For example, during jEdit 4.1-4.2, a developer removed two methods from the `FoldVisibilityManager` class and added a similar method in the `DisplayManager` class; five months later the developer removed another method from the `FoldVisibilityManager` class and added a similar method in the `DisplayManager` class and finally made the `FoldVisibilityManager` class replaced by `DisplayManager`. This finding implies that a large part of systematic changes cannot be identified by time window based evolution analysis [20], which treats code changes in the same time window (e.g., the same commit or the same day) as one transaction. Trajectory patterns, which do not depend on time window analysis, may be used to identify systematic changes that span over a long time.

Table V. Span Analysis Results

System	Period	Month	#Day			#File			#Developer		
			Max	Med	Avg	Max	Med	Avg	Max	Med	Avg
jEdit	3.0-3.1	4	86	9	24.89	2	1	1.14	1	1	1
	3.1-3.2	4.2	117	4	23.55	4	1	1.41	1	1	1
	3.2-4.0	7.5	221	30.5	62.95	6	1	1.55	1	1	1
	4.0-4.1	10.6	282	10	58.73	5	1	1.40	2	1	1.04
	4.1-4.2	18	507	100	193.15	37	1	7.68	2	1	1.17
jBPM	5.1-5.2	5.6	204	1	37.56	7	1	1.47	3	1	1.34
	5.2-5.3	5	120	13	23.04	9	2	2.69	3	1	1.62
	5.3-5.4	6	153	7	20.10	8	1	1.51	3	1	1.19
SWT	3.5-3.6	12	268	14	59.06	14	3	2.40	5	1	1.76
	3.6-3.7	12.5	316	28	60.01	5	1	1.43	4	1	1.58

5.4. RQ3: Impact of Thresholds

Our trajectory pattern identification algorithm depends on the following thresholds: $threshold_{frag}$ (minimum length of a code fragment by character when identifying similar methods or code fragments), $threshold_{prop}$ (minimum number of common properties of a group), $threshold_{minIns}$ (minimum number of instances in a group), $threshold_{maxIns}$ (maximum number of instances in a group). To investigate the impact of a threshold on the identification of trajectory patterns, we compared the identified trajectory patterns under different values of each individual threshold for each subject system with the following baseline settings: $threshold_{frag} = 100$, $threshold_{prop} = 2$, $threshold_{minIns} = 2$, $threshold_{maxIns} = 10$. When analyzing a threshold, the other three thresholds took the values of their baseline settings.

The impact of $threshold_{frag}$ on different types of trajectory patterns in the three subject systems is shown in Figure 14 with Y axis in log scale. Generally, the increase of $threshold_{frag}$ will make less trajectory patterns be identified, since it will reduce the number of change group associations of *Similar Content*. From Figure 14, it can be seen that the impact of $threshold_{frag}$ is very small. $threshold_{frag}$ is used to filter out code fragments or methods that are not big enough when identifying similar content for change grouping or change group association. It mainly impacts the identification of Type-6 patterns, as they are usually related to adding or removing methods with similar content. After analyzing the data, we found that the average length of added or removed methods involved in Type-6 patterns is about 1170 characters with a median of 583 characters. In other words, most of these methods are longer than 300 characters. Therefore, the number of identified Type-6 patterns only declines slightly when $threshold_{frag}$ increases from 50 to 300.

The impact of $threshold_{prop}$ on different types of trajectory patterns in the three subject systems is shown in Figure 15 with Y axis in log scale. It can be seen that $threshold_{prop}$ has a great impact on the identification of trajectory patterns. The number of identified trajectory patterns declines

DOI: 10.1002/smr

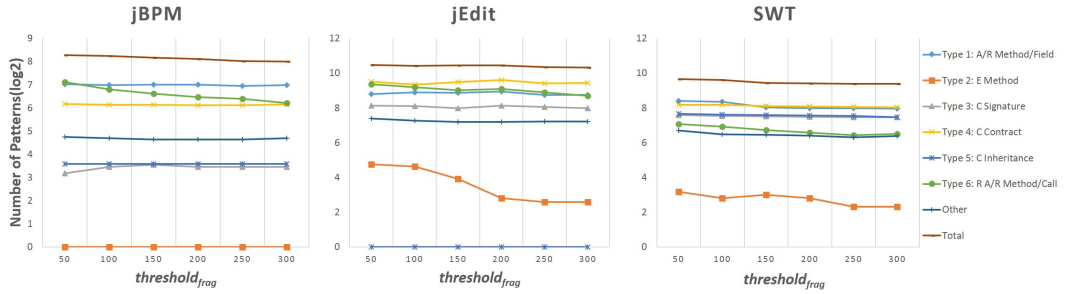

Figure 14. Impact of $threshold_{frag}$ on Identification of Trajectory Patterns

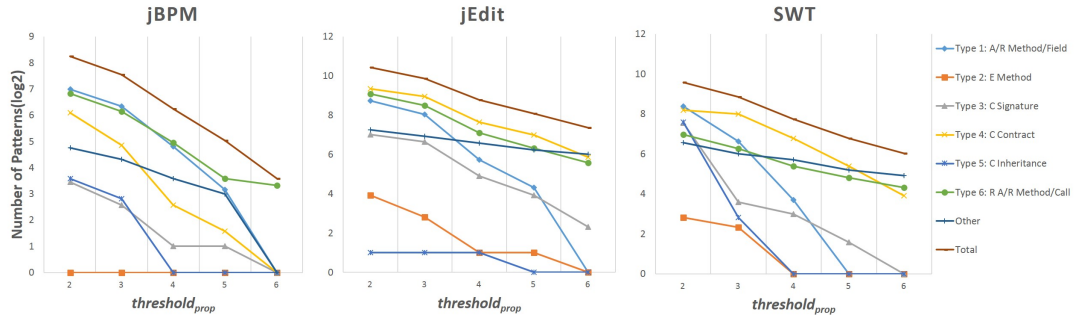

Figure 15. Impact of $threshold_{prop}$ on Identification of Trajectory Patterns

significantly with the increase of $threshold_{prop}$. Only a few trajectory patterns can be identified if $threshold_{prop}$ is set to 5 or larger. For example, when $threshold_{prop}$ is set to 6, the numbers of trajectory patterns in jBPM, jEdit and SWT are 12, 163 and 65, respectively. In other words, most of the identified trajectory patterns have at least one change group possessing only two to four common properties. The impact of $threshold_{prop}$ in jBPM is a little different from its impact in the other systems. When $threshold_{prop}$ is set to 5, nearly no trajectory patterns can be identified in jBPM but some can still be identified in jEdit and SWT. After analyzing the data, we found that the trajectory patterns identified in jEdit and SWT are mainly Type-4 and Type-6 patterns. These patterns usually have a number of common properties, including several method calls (*context*) in addition to the common *class* and *developer*.

The impact of $threshold_{minIns}$ on different types of trajectory patterns in the three subject systems is shown in Figure 16. It can be seen that $threshold_{minIns}$ has a great impact on the identification of trajectory patterns. The number of identified trajectory patterns declines significantly with the increase of $threshold_{minIns}$. Only a few trajectory patterns can be identified if $threshold_{minIns}$ is set to 6 or larger. In other words, most of the identified trajectory patterns have at least one change group having only two to five change operations. An exception is that the number of Type-4 trajectory patterns increases when $threshold_{minIns}$ increases from two to three in jBPM. After analyzing the data, we found that, in this case the number of the identified change groups declines with the increase of $threshold_{minIns}$, which in turn makes some patterns be split into multiple smaller ones thus causes the increase of identified patterns. We observed that in SWT when $threshold_{minIns}$ increases from 2 to 3 the number of Type-5 patterns declines from about 200 to near 0. Type-5 patterns are related to change of inheritance hierarchy. Change groups involved in Type-5 patterns are usually about adding or removing class, adding or removing inheritance, or changing inheritance. In SWT this kind of change groups usually involve only two change operations related to two classes, so when $threshold_{minIns}$ increases from 2 to 3 most of these change groups are filtered out. We also observed that the increase of $threshold_{minIns}$ has a much smaller impact on trajectory patterns of the type *Other*. After analyzing the data, we found that most of the change groups involved in this kind of patterns are about method calls (Add/Remove/Path

DOI: 10.1002/smr

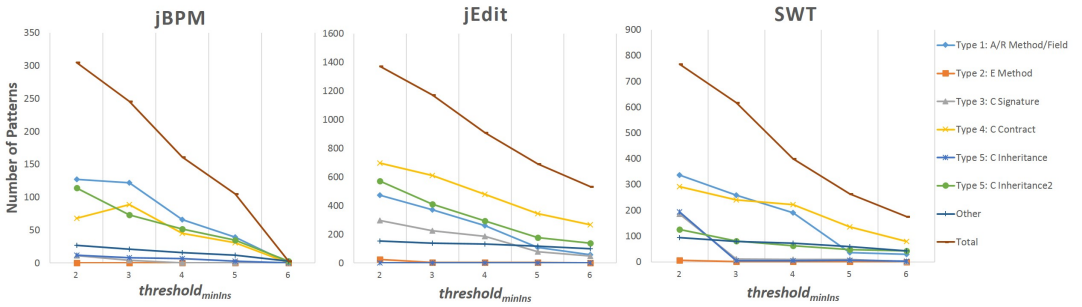

Figure 16. Impact of $threshold_{minIns}$ on Identification of Trajectory Patterns

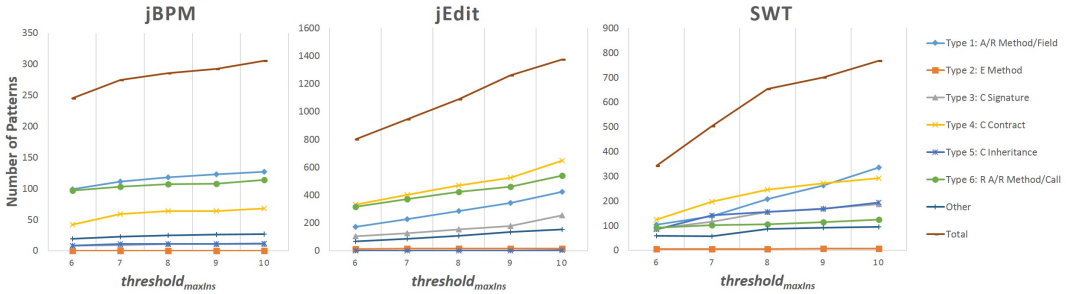

Figure 17. Impact of $threshold_{maxIns}$ on Identification of Trajectory Patterns

Change) and each of them may involve a number of (greater than 6) method calls in the same big method as its instances (change operations). So the number of trajectory patterns of the type *Other* only declines slightly when $threshold_{minIns}$ increases from 2 to 6.

The impact of $threshold_{maxIns}$ on different types of trajectory patterns in the three subject systems is shown in Figure 17. It can be seen that $threshold_{maxIns}$ has a significant impact on the identification of trajectory patterns. The number of identified trajectory patterns increases significantly with the increase of $threshold_{maxIns}$. It can be seen that the change of $threshold_{maxIns}$ has a smaller impact on jBPM. After analyzing the data, we found that the maximum number of instances in the change groups of most of the patterns identified in jBPM is less than eight, while the number in *jEdit* and *SWT* is evenly distributed between six and ten. Another finding is that, when $threshold_{maxIns}$ is set to 11 or larger, a large part of the newfound patterns consist of trivial change groups with only two common properties (the same class and the same developer).

In summary, all the four thresholds have impact on the identified trajectory patterns and $threshold_{prop}$ and $threshold_{minIns}$ are two key thresholds among them. The impact of $threshold_{frag}$ is small. Considering the typical length of added or removed code fragments, a proper setting of $threshold_{frag}$ may be between 100 and 300. As there are few trajectory patterns possessing five or more common properties for each change group, a proper setting of $threshold_{prop}$ may be between 2 and 4. And as there are few trajectory patterns possessing six or more change operations for each change group, a proper setting of $threshold_{minIns}$ may be between 2 and 5. A larger setting of $threshold_{maxIns}$ can significantly increase the number of identified trajectory patterns, but may introduce a lot of patterns with trivial commonalities and significantly increase the computation overhead. We think $threshold_{maxIns} = 10$ is a proper setting for most systems, but a larger setting (e.g., 15 or more) can also be used if the computation overhead is acceptable.

5.5. RQ4: Rules and Problems Revealed by Trajectory Patterns

After analyzing the results, we found that the identified trajectory patterns can reveal underlying rules and problems about the evolution process from several aspects: *Team Convention*

DOI: 10.1002/smr

reflects conventional development schedules that are followed by different developers of the project; *Personal Habit* represents developers' personal habits in individual development tasks; *Collaboration Mode* describes how different developers collaborate in a task; *Exception* means exceptions to an identified pattern, which may indicate suspicious changes. These rules and problems can be revealed from corresponding trajectory patterns by analyzing the time orders and corresponding developers of involved change operations.

1) *Team Convention*: Team convention can be identified by considering the order of different change groups in a trajectory pattern. For example, the pattern shown in Figure 12 indicates the following convention: after a set of business methods are added, corresponding test cases are developed by adding method calls to these new methods in existing test methods; if the tests fail, corresponding method calls will be removed from the test methods.

2) *Personal Habit*: Personal habit can be identified by considering the order of a developer's commits involved in a trajectory pattern. For example, the pattern shown in Figure 8 indicates that the developer is used to revise one method signature and corresponding method calls in a commit and then revise the next in another commit. Some other patterns indicate that another developer is used to revise multiple method signatures and corresponding method calls in one commit.

3) *Collaboration Mode*: Collaboration mode can be identified by considering how different developers were involved in a trajectory pattern. A trajectory pattern involving change groups by different developers usually indicates collaboration by different types of development tasks. For example, the pattern shown in Figure 12 indicates that a developer implemented new functionalities by adding a series of methods and another developer fixed the bugs in the methods and confirmed the revision by unit testing. On the other hand, a change group involving change operations by different developers usually indicates collaboration by different parts of the system. For example, a group of adding method operations by two developers usually indicates that they are responsible for different modules.

4) *Exception*: Exceptions to a trajectory pattern can be identified from the association aspect. A change operation that is in a change group but is not involved in an association with another group is an exception. For example, the pattern shown in Figure 11 indicates that one of the added methods is not involved in the *Same Method* (a special case of *Similar Content*) association with the other group. This may indicate an incomplete "Pull Up Method" refactoring.

The numbers of rules and problems revealed by trajectory patterns are shown in Table VI. As jEdit involved only one developer during that time, there are no *Team Convention* patterns or *Collaboration Mode* patterns identified from 3.0 to 4.0. We identified 308 *Team Convention* patterns. Typical examples include: 1) first defining interfaces then developing test cases and after that implementing the interfaces; 2) first defining interfaces then implementing them and adding method calls to them; 3) first changing methods then defining test cases and adding method calls to these methods to test them. We identified 1,991 *Personal Habit* patterns. These patterns reflect that, some developers like to accomplish a development task in two or more commits, while some others like to accomplish a task in one commit. We identified 451 *Collaboration Mode* patterns. These patterns reflect different kinds of collaboration modes among two or more developers, for example, one developer implements a functionality and another one fixes bugs in the implementation. We identified 83 *Exception* patterns. These patterns usually reflect incomplete or inconsistent code changes.

The rules and problems revealed by trajectory patterns can facilitate personal and team process improvement and quality assurance. A team leader or developer can collect and analyze change-related process data by trajectory patterns to identify improvement opportunities in terms of team convention, personal habit, and collaboration mode. Based on *Team Convention* patterns, a team leader can make better development schedule and achieve more accurate progress measurement based on known team conventions. Based on *Personal Habit* patterns, a developer can monitor his/her own development routines and analyze what kind of personal habits best fits himself/herself in terms of development efficiency and quality. Based on *Collaboration Mode* patterns, a team leader can know the role of each developer in collaboration and thus can make better decisions of

DOI: 10.1002/smr

Table VI. Numbers of Rules and Problems Revealed by Trajectory Patterns

System	Period	*Team Convention*	*Personal Habit*	*Collaboration Mode*	*Exception*
jEdit	3.0-3.1	0	65	0	1
	3.1-3.2	0	132	0	3
	3.2-4.0	0	377	0	8
	4.0-4.1	6	222	9	5
	4.1-4.2	74	471	98	20
jBPM	5.1-5.2	9	57	14	2
	5.2-5.3	31	91	44	11
	5.3-5.4	11	78	17	5
SWT	3.5-3.6	135	373	204	21
	3.6-3.7	42	125	65	7

task assignment. Based on *Exception* patterns, quality assurance personnel can identify suspicious code changes such as violation of procedures, incomplete refactoring, and even bugs.

5.6. RQ5: Evolution Analysis of Trajectory Patterns

Trajectory patterns reveal systematic and collaborative code changes during continuous evolution. On the other hand, trajectory patterns themselves evolve over time. For example, new instances (including change operations and association instances) may be added to existing trajectory patterns; existing trajectory patterns may be extended with new change groups and change group associations. Analyzing the evolution of trajectory patterns cannot only help understand the forming process of trajectory patterns, but also reveal additional rules and problems of the evolution process.

To this end, we analyzed the evolution of trajectory patterns across different periods (see the periods shown in Table IV) for each of the three subject systems. For each period, we ran SETGA to identify trajectory patterns from all the commits in the period and all the periods before it. For example, when analyzing the evolution for the period jEdit 3.2-4.0, all the commits in the first three periods of jEdit (i.e., 3.0-3.1, 3.1-3.2, 3.2-4.0) were considered for trajectory pattern identification. The evolution analysis was conducted with the following threshold settings: $threshold_{frag} = 100$, $threshold_{prop} = 3$, $threshold_{minIns} = 2$, $threshold_{maxIns} = 10$.

The evolution trends of trajectory patterns in jBPM, jEdit, and SWT are shown in Figure 18, Figure 19, and Figure 20 respectively. Each figure shows the changes of the numbers of identified trajectory patterns and instances over different periods. The number of instances means the number of change operations in all the identified trajectory patterns. Each figure also shows increase the number of lines of code over different periods.

It can be seen that the numbers of identified trajectory patterns and instances grow over time. In some periods, the numbers grow more rapidly than in other periods. This is mainly due to the changes of development tasks, commit frequency, and developers. For example, the numbers of trajectory patterns and instances identified in jBPM rapidly grow from the period 5.2-5.3. After analyzing the commit logs, we found that during the period 5.1-5.2 a large part of the commits are dedicated on updating various kinds of documents such as XML files and comments in source files. From the period 5.2-5.3 the commits are more dedicated on implementing new functionalities and fixing bugs. Moreover, code changes during the period 5.1-5.2 were scattered in more packages, so we guess that the development tasks in the period 5.1-5.2 were less systematic and thus the identified trajectory patterns were less than those in the following periods.

The growths in the numbers of identified trajectory patterns and instances are closely related to the increase in the number of code base sizes. But the numbers of identified trajectory patterns and instances are increasing faster than the size of the code in jBPM 5.3-5.4. After viewing the codes and log messages, we found that the most development tasks in the period 5.3-5.4 are fixing bugs or updating the existing functionalities, and the development tasks are closely related to development tasks in period 5.2-5.3. For example, the developers added the new functionalities in the period 5.2-5.3 and fixted or updated the functionalities in period 5.3-5.4.

We further analyzed the changes of individual trajectory patterns and their instances along with the periods. Based on the analysis, we found the following typical evolution types of trajectory patterns, each of which reveals some additional rules or problems of the evolution process. In all the

DOI: 10.1002/smr

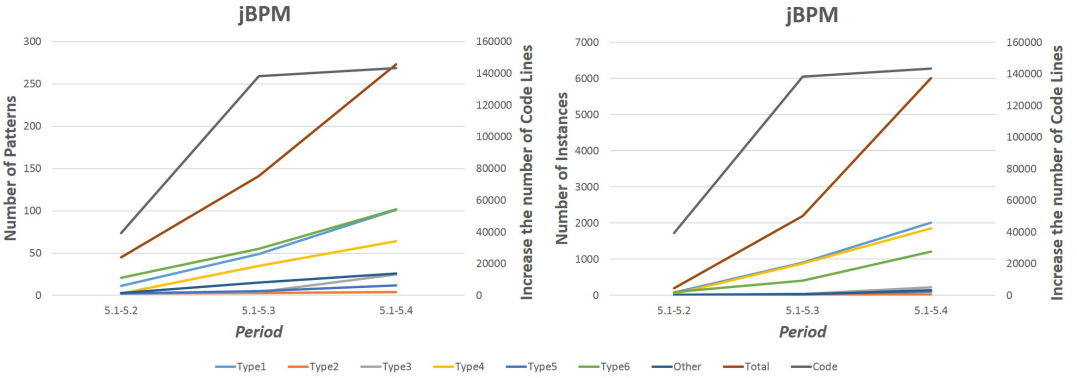

Figure 18. Trajectory Pattern Evolution Trend in jBPM

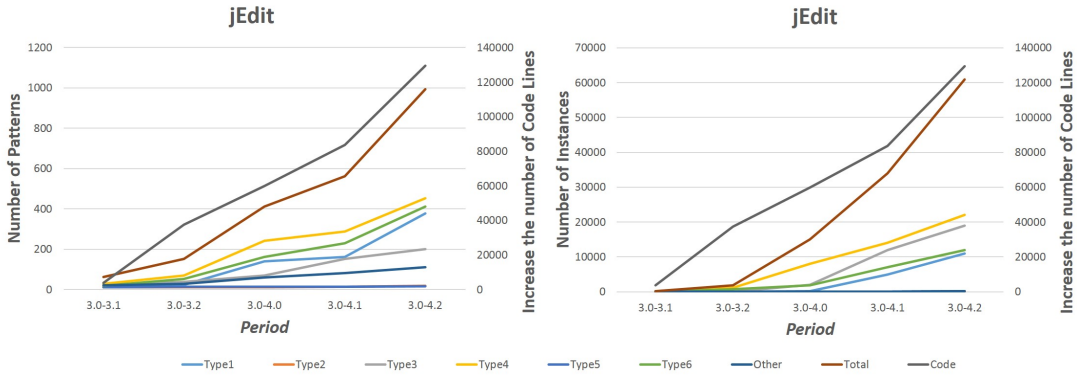

Figure 19. Trajectory Pattern Evolution Trend in jEdit

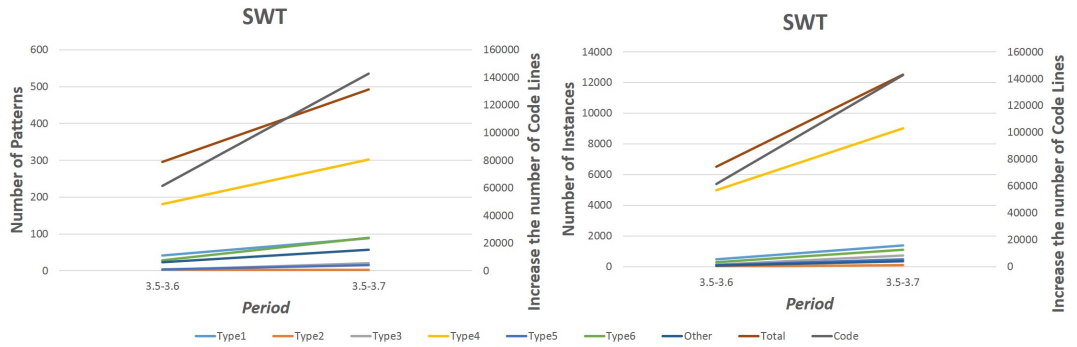

Figure 20. Trajectory Pattern Evolution Trend in SWT

figures in this part, the part with gray background represents the extension after evolution, and the part in the dotted box represents the portion that is eliminated after evolution.

1) *Pattern Extension*: Pattern extension means that an existing trajectory pattern is extended with new change groups and change group associations. Pattern extension often reflects a systematic and gradual implementation process of a development task such as adding new features and refactoring. During the process, new functionalities may be gradually added, bugs may be identified and fixed, and design may be incrementally improved. An example of pattern extension in SWT is shown in Figure 21. In the original pattern, to fix the issue "MouseWheel event does not contain

DOI: 10.1002/smr

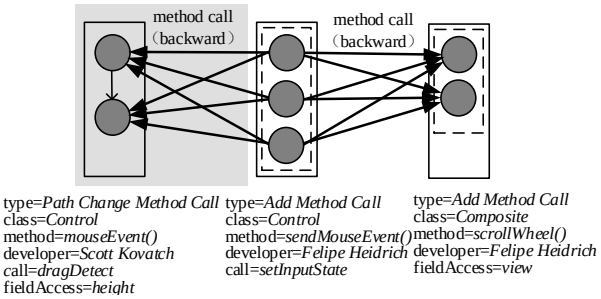

Figure 21. An Example of Pattern Extension in SWT

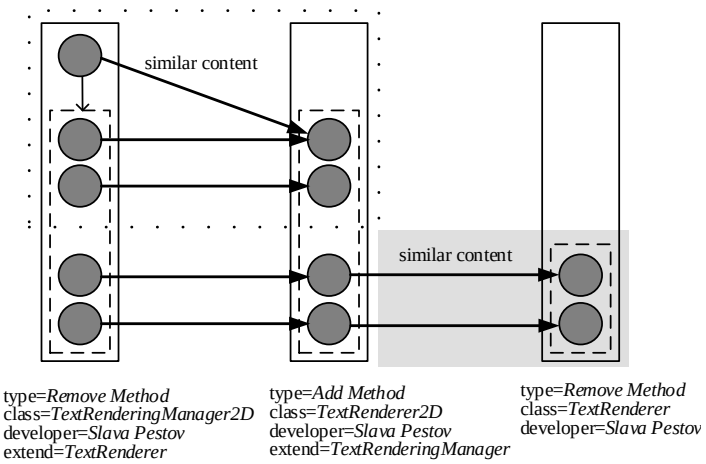

Figure 22. An Example of Pattern Extension in jEdit

information on the axis", a developer *Felipe Heidrich* added three method calls in the method `sendMouseEvent` of the class `Control` and added two method calls to `sendMouseEvent` in the method `scrollWheel` of the class `Composite`. In the following period, to implement deferring double click events for two controls, another developer *Scott Kovatch* changed the path of the method call to `sendMouseEvent` in the method `mouseEvent` of the class `Control`; to fix the issue "click count doesn't reset if multiple buttons are clicked", he changed the path of the method call again. Another example of pattern extension in jEdit is shown in Figure 22. In the original pattern, to improve the design a developer *Slava Pestov* conducted a refactoring by moving five methods from the class `TextRenderingManager2D` to the class `TextRenderer2D`, among which two similar methods were merged. In the following period, the developer went on the refactoring by removing two methods from the class `TextRenderer` that are similar to two of the methods moved from `TextRenderingManager2D` to `TextRenderer2D`. Note that in this example the original pattern will be kept since not all of its instances are involved in the new pattern.

2) *Pattern Exception Repairing*: Pattern exception repairing means that some exceptions in an existing trajectory pattern are repaired. In other words, some change operations in an existing pattern are added to some change group associations which they are not involved in originally. Pattern exception repairing often implies bug fixing that eliminates inconsistent code changes. An example of pattern exception repairing in jBPM is shown in Figure 23. In the original pattern, a developer *salaboy* added method calls to `getCompleted` and `size` in two test methods `testLifeCycle` and `testLifeCycleMultipleTasks` of the class `TaskLifeCycleBaseSyncTest`. As the test was finished, *krisv* removed the two method calls in `testLifeCycle`, but forgot to remove the method calls in `testLifeCycleMultipleTasks`. In the next period, to fix the test *krisv* removed the method calls in `testLifeCycleMultipleTasks`. Another example of

DOI: 10.1002/smr

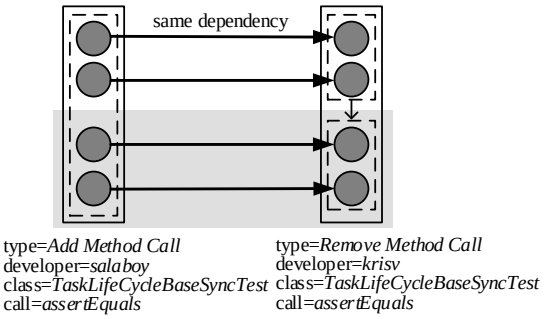

Figure 23. An Example of Pattern Exception Repairing in jBPM

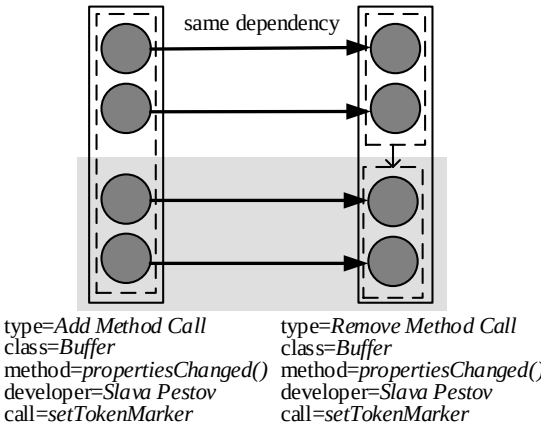

Figure 24. An Example of Pattern Exception Repairing in jEdit

pattern exception repairing in jEdit is shown in Figure 24. In the original pattern, to add explicit folding a developer *Slava Pestov* added four method calls to `getProperty`, `putProperty`, `getProperty` and `putProperty` respectively in the method `propertiesChanged` of the class `Buffer`. After that, to implement more folding manager work, he removed method calls to `getProperty` and `putProperty` but forgot to remove the other two. In the next period, to fix folding manager bugs, he removed method calls to `getProperty` and `putProperty`.

3) *Pattern Reuse*: Pattern reuse means that new change operations and corresponding association instances are added to an existing trajectory pattern. Pattern reuse often indicates that existing trajectory patterns are followed and applied in new development tasks. An example of pattern reuse in jEdit is shown in Figure 25. In the original pattern, a developer *Slava Pestov* first added a parameter to the method `setEntry` of the class `BufferHistory`, then added method calls to it in two methods `closeAllBuffers` and `_closeBuffer` of the class `jEdit`. In the next period, he added another parameter to the method `setEntry` of the class `BufferHistory`, then added method calls to the new method in two methods `closeAllBuffers` and `_closeBuffer` of the class `jEdit`.

The results of pattern evolution analysis are shown in Table VII. We only found two *Pattern Exception Repairing* instances from jEdit and jBPM. For each system, we found dozens of *Pattern Extension* instances and *Pattern Reuse* instances. The distributions of different types of trajectory patterns in pattern extension and pattern reuse are shown in Figure 27 and Figure 28 respectively. They show the percentage of each type of patterns in all the pattern extension or reuse instances identified in different projects. It can be seen that Type-1 (*Add/Remove Method/Field*), Type-4 (*Change Method Contract*), and Type-6 (*Repeated Adding/Removing Methods/Method Calls*)

DOI: 10.1002/smr

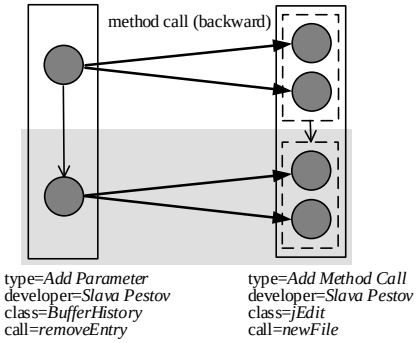

Figure 25. An Example of Pattern Reuse in jEdit

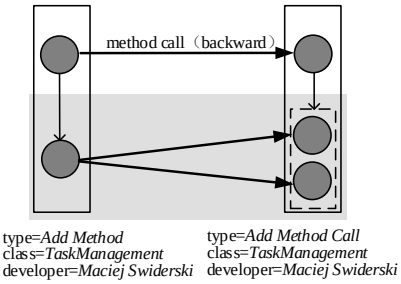

Figure 26. An Example of Pattern Reuse in jBPM

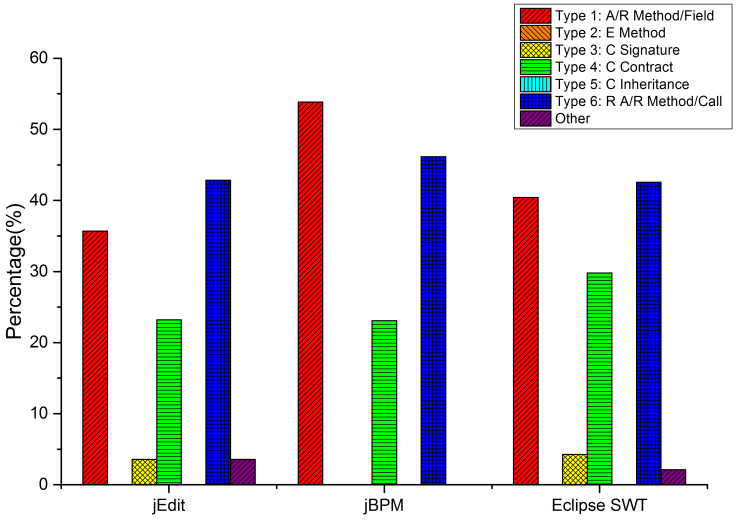

Figure 27. Distribution of Different Types of Trajectory Patterns in Pattern Extension

patterns are more likely to be extended and reused. These three types of patterns are also more popular than the other types.

Table VII. Pattern Evolution Analysis Results

System	*Pattern Extension*	*Pattern Exception Repairing*	*Pattern Resuse*
jEdit	56	1	101
jBPM	13	1	28
SWT	47	0	75

DOI: 10.1002/smr

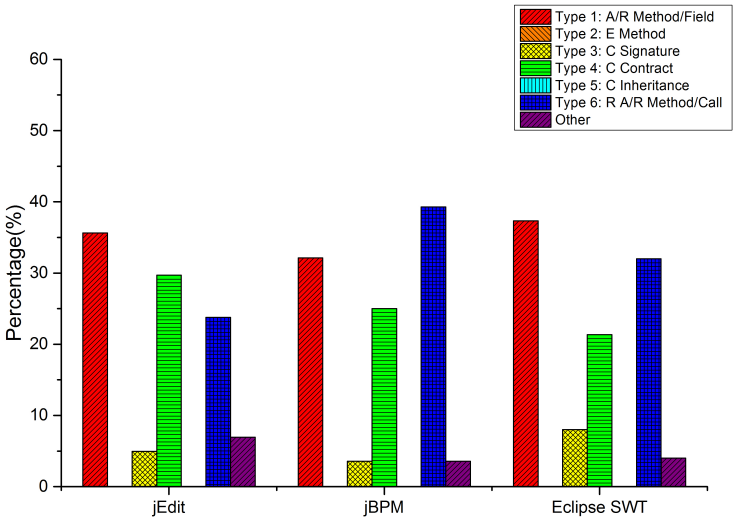

Figure 28. Distribution of Different Types of Trajectory Patterns in Pattern Reuse

5.7. Threats to Validity

A threat to the internal validity of our empirical study lies in the thresholds used in SETGA. To eliminate meaningless change groups and ensure acceptable performance, SETGA uses a threshold $threshold_{maxIns}$ to limit the maximum number of instances in a group. This threshold, however, may eliminate meaningful trajectory patterns that involve a large number of similar code changes such as a large-scale refactoring involving dozens of similar extracting method changes. For this threat, we have studied the impact of different thresholds on the identification of different kinds of trajectory patterns (see Section 5.4). Another threat to the internal validity is the interpretation of the mined trajectory patterns and their evolution. Currently, we manually interpreted the meaning of mined patterns and their evolution by understanding the intentions of related code changes by reading code, emails, commit messages, and related issue reports. This kind of interpretation may lead to misunderstanding or incorrect categorization of systematic changes. The other threat to the internal validity lies in branch merging in Git repositories. Although we only analyzed the commit records from the master branch of a Git repository, there may be merged commits in the master branch because of simultaneous parallel development. This means that an evolutionary trajectory pattern mined from the commits may involve code changes from different teams. To alleviate this threat, we tried to choose evolution periods that contain as few branch merging as possible in our study.

The threats to the external validity of our empirical study mainly lie in the fact that we only conducted empirical study on three open-source systems. All our findings may not be applicable to industrial projects with hundreds or thousands of developers. It is likely that these projects may demonstrate quite different trajectory patterns compared with open-source projects due to their differences on team organization and process management. To lower the impact of this threat, we have made our tool available online to allow other researchers to extend our study on other kinds of projects.

6. DISCUSSION

SETGA combines two kinds of abstraction mechanisms to provide a high-level overview of the evolutionary trajectory of software systems. **Generalization** is used to identify common properties and associations shared by a group of change operations. **Aggregation** is used to combine potentially relevant change groups together based on various kinds of associations. As such, SETGA

DOI: 10.1002/smr

is capable of abstracting a series of code changes dispersed in different locations and at different times.

A fundamental assumption of SETGA is that change operations occurring in relevant program units (e.g., the client and supplier sides of a method call) or with similar content are potentially relevant. This kind of relevant code changes can be aggregated to recover systematic and collaborative high-level changes that are not explicitly documented. Although individual associations between code changes are not necessarily definitive evidence for the relevance, we believe that different kinds of code changes are likely to be relevant if we can identify groups of code changes with similar associations.

From Table V, it can be seen that the trajectory patterns identified by SETGA usually involve only one to several files. We believe that it is due to the fact that the aggregation of different change groups depends on explicit and direct associations such as method calls and inheritance. It is possible that more trajectory patterns can be identified if more association types are considered. To achieve a more comprehensive recognition of potentially relevant code changes, it is useful to incorporate more sophisticated program analysis techniques such as data dependency and control dependency analysis used in change impact analysis.

Identification and analysis of trajectory patterns can facilitate software evolution management and quality assurance in several ways. First, exceptions to an identified trajectory pattern can indicate possible violations of development conventions. For example, a developer may violate a "test first" convention by adding new classes to a business package without creating corresponding test cases first. Quality assurance personnel thus can intervene as soon as such exceptions are detected. Second, identified trajectory patterns can be used to guide development task assignment and scheduling. For example, past trajectory patterns may indicate optimal time sequence and developer assignment that can lead to high-quality release in specific context. Third, there may exist correlations between trajectory patterns and bugs. For example, a series of new methods introduced by a novice developer without a follow-up refactoring by an experienced developer may be likely to introduce bugs. Change-base bug prediction predicts the introduction of bugs in code changes by training machine learning classifiers on software history [16]. Based on trajectory patterns, we can train classifiers and predict bugs by systematic changes revealed by trajectory patterns instead of individual code changes. We can extract features such as the numbers of different types of change operations and their orders for classifier training. As a more logically complete unit of code changes, it is likely that the trained classifiers can achieve better performance (e.g., precision and recall) in change-based bug prediction.

SETGA provides high-level overview of software evolution by abstracting a series of code changes as a trajectory pattern. This kind of overview, however, is limited, as SETGA may produce hundreds of trajectory patterns for a single release. To provide more comprehensive evolution overview, we need to further summarize identified trajectory patterns. This can be done from the following aspects. First, as discussed above, we can incorporate more association types in the identification of trajectory patterns by employing more sophisticated program analysis techniques. Second, we can construct a hierarchical categorization of trajectory patterns by using hierarchical clustering. For example, trajectory patterns involving code changes in the same modules/packages or with similar topics can be clustered together at different levels, so as to provide evolution overview at different levels. Third, we can combine text analysis and natural language processing to generate text summarization for identified trajectory patterns by utilizing the text content of code changes, commit messages and related issues.

Currently, our definition and mining algorithms of trajectory patterns do not support the aggregation of related changes in parallel branches. Nor do they support the merging of code changes of different branches. And in our empirical study, we only analyzed the commit records from the master branches of Git repositories. Another limitation lies in the treatment of renaming of classes and methods. Currently SETGA treats a renaming change as a combination of removing and adding class/method. This limitation may make SETGA omit some refactoring-related trajectory patterns and produce useless patterns.

DOI: 10.1002/smr

7. CONCLUSION

In this paper, we have proposed the concept of trajectory pattern that groups and aggregates relevant code changes made at different times and in different locations. We have presented SETGA, an approach that can identify trajectory patterns from historical commit records from version control systems. The proposed approach has been implemented and applied to three open-source systems. The results show that SETGA can identify various types of trajectory patterns that can reflect the evolutionary trajectory over time and reveal underlying rules and problems about the evolution process.

Our future work will improve our approach and tool from three aspects. First, we will use issue IDs in commit messages to link code changes to issues and consider the associations (e.g., code changes for the same issue) implied by these links in the identification of trajectory patterns. Second, we will combine text analysis to utilize the text content of code changes, commit messages and related issues in the identification and interpretation of trajectory patterns. Third, we will further investigate the correlation between different kinds of trajectory patterns and software development quality/efficiency. Based on the improvement of the approach and tool, we will conduct more case studies with industrial and open-source systems to explore how SETGA can be applied in software evolution management and quality assurance.

ACKNOWLEDGEMENT

This work is supported by National Key Research and Development Program of China under Grant No. 2016YFB1000800, National Natural Science Foundation of China under Grant No. 61370079, National High Technology Development 863 Program of China under Grant No. 2012AA011202.

REFERENCES

1. F. Beat, W. Michael, P. Martin, and G. Harald C. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov. 2007.
2. D. Beyer and A. E. Hassan. Animated visualization of software history using evolution storyboards. In *Working Conference on Reverse Engineering*, pages 199–210, 2006.
3. S. Bouktif, Y.-G. Gueheneuc, and G. Antoniol. Extracting change-patterns from cvs repositories. In *Working Conference on Reverse Engineering*, pages 221–230, 2006.
4. C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *ACM symposium on Software visualization*, pages 77–86, 2003.
5. M. D'Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering*, 35(5):720–735, Sept. 2009.
6. H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *International Conference on Software Maintenance*, pages 99–108, 1999.
7. Q. Jiang, X. Peng, H. Wang, Z. Xing, and W. Zhao. Summarizing evolutionary trajectory by grouping and aggregating relevant code changes. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 361–370, 2015.
8. H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, Mar. 2007.
9. M. Kim and D. Notkin. Discovering and representing systematic code changes. In *International Conference on Software Engineering*, pages 309–319, 2009.
10. M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *International Conference on Software Engineering*, pages 333–343, 2007.
11. M. Kim, D. Notkin, D. Grossman, and G. Wilson Jr. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45–62, Jan. 2013.
12. A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.
13. S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *International Conference on Software Engineering*, 2014.
14. 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 *Automated Software Engineering*, pages 180–190, 2013.
15. W. Qian, X. Peng, Z. Xing, S. Jarzabek, and W. Zhao. Mining logical clones in software: Revealing high-level business and programming rules. In *International Conference on Software Maintenance*, pages 40–49, 2013.
16. S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.

DOI: 10.1002/smr

17. F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *International Conference on Software Maintenance*, pages 328–337, 2004.
18. Z. Xing and E. Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31(10):850–868, Oct. 2005.
19. A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, Sept. 2004.
20. T. Zimmermann and P. Weigerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, May 2004.
21. T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

DOI: 10.1002/smr