Abstract—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 locations of the changes. After that, SETGA aggregates relevant change operation groups by mining various associations among them. 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 are useful for software evolution management and quality assurance.

Index Terms—Evolution, Version Control System, Code Change, Pattern, Mining.

I. 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.

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 [1]. 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 [2], [3], [4] and fine-grained repetitive code changes [5], [6]. These approaches can only relate code changes within the same transactions or time windows. There also have been some approaches [7], [8], [1] 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 a case 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 underlying evolution rules and problems that can be
revealed by the identified patterns. The results demonstrate
the usefulness and effectiveness of SETGA for summarizing
evolutionary trajectory for software maintenance.

The rest of the paper is structured as follows. Section II
reviews some existing proposals and compares them with
ours. Section III defines trajectory pattern and other related
concepts. Section IV describes the proposed approach. Section
V evaluates the proposed approach with a case study. Section
VI discusses some related issues. Section VII concludes the
paper and outlines the future work.

II. RELATED WORK

One of the main purposes of mining software repositories
(MSR) is to analyze trends and recurring patterns of software
changes. A comprehensive literature survey on approaches
for mining software repositories in the context of software
evolution is presented in [9].

Some approaches aim to analyze evolution phases and styles
of software evolution. Xing and Strouli [10] proposed an ap-
proach 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. [11] presented a three-
dimensional visual representation for examining a system’s
release history. Collberg et al. [12] presented a visualization
system that extracts evolution information from version control
systems and displays it using a temporal graph visualizer. Van
RysSELBERGHE and DEMeyer [13] applied a simple visualization
technique to recognize relevant changes such as unstable
components and coherent entities. Beyer and Hassan [14]
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. [15] 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. [2] applied data
mining techniques on the revision history of a system to detect
source files that are usually changed together. Zimmermann
et al. [3] proposed an approach that mines association rules
among the changes of program entities such as functions or
variables. Bouktif et al. [4] 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. [5] extracted method-level code
changes by comparing abstract syntax trees (ASTs) of two

III. DEFINITION

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

A. Change Operation

A change operation represents an atomic code change (e.g.,
adding/removing code fragment, adding/removing method pa-
rameter) as well as its evolution properties (e.g., time, de-
veloper, location). Evolution properties of change operations
provide the basis for change operation grouping. The concept
of change operation is defined as follows.

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

According to Table I, a change operation can be adding/re-
moving/changing a class, field or method. For a class,
the change can be adding/removing itself or adding/remov-
ing/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. For a method
body, the change can be adding/removing its code fragments
or adding/removing/changing field accesses or method calls. 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.

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, 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.

B. 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 are similar, or the added or removed code fragments of two adding/removing code change operations are similar.

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.

C. Trajectory Pattern

Potentially relevant change groups can be further aggregated by various associations between individual change operations. The association of two change operations ch1 and ch2 satisfying that ch1 is committed before or at the same time (in the same commit) of ch2 can be represented by $ch_1 \rightarrow_{type} ch_2$, where type is one of the following association types:

- Method Call (forward): ch2 is a change about the method or method body level and ch1 is a change about the method call to ch2’s target method (Add/Remove/Path Change);
- Method Call (backward): ch1 is a change at the method or method body level and ch2 is a change about the method call to ch1’s target method (Add/Remove/Path Change);
- Field Access (forward): ch2 is a change about a field and ch1 is a change about the access to the field (Add/Remove/Path Change);
- Field Access (backward): ch1 is a change about a field and ch2 is a change about the access to the field (Add/Remove/Path Change);
- Inheritance/Interface Implementation (forward): ch1 and ch2 are both class-level changes and ch1’s target class inherits (implements) ch2’s target class (interface) before or after the change;
- Inheritance/Interface Implementation (backward): ch1 and ch2 are both class-level changes and ch2’s target class inherits (implements) ch1’s target class (interface) before or after the change;
- Same Dependency: ch1 and ch2 are both changes about method call/field access but of different change types (e.g., one is adding and the other is removing) and the called method/accessed field are the same;
- Similar Content: ch1 and ch2 are both changes about adding/removing methods or code fragments but of different change types (e.g., one is adding and the other is removing) and the methods or code fragments are similar (a special case is that ch1 and ch2 are changes about the same methods or code fragments).

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.
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 \( \text{threshold}_{\text{frag}} \), a change operation is generated with its source code as the change content.

### B. 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 [16] 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 previous work [17] 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 \( \text{ChOpSet} \). It returns a set of change groups \( \text{Groups} \), each of which consists of change operations of the same type and sharing some common properties. For each change operation \( \text{change} \) in \( \text{ChOpSet} \), the function \( \text{prop} \) returns a set of properties of \( \text{change} \) that can be considered when computing the common properties with other change operations (see Section III-B).

Three predefined thresholds are used in the algorithm: \( \text{threshold}_{\text{prop}} \) specifies the minimum number of common properties of a group; \( \text{threshold}_{\text{minIns}} \) specifies the minimum number of instances (change operations) in a group; \( \text{threshold}_{\text{maxIns}} \) 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 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).

The algorithm first generates all the binary associations between change groups in Groups (Line 3-14). For each G, G’ ∈ Groups, the algorithm identifies possible change group associations from G to G’ using the associateGroups(G, G’) function (See Algorithm 3) (Line 5). As there may be multiple types of associations between two change groups, associateGroups(G, G’) returns a set of change group associations. 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 with each binary association asso in BinAssos (Line 18-26). The extendAggregation(agg, asso) function (See Algorithm 4) 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). 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).

C. Change Group Aggregation

SETGA uses a change group aggregation algorithm (See Algorithm 2) to identify trajectory patterns. The algorithm takes as input a set of change groups Groups and returns a set of trajectory patterns (i.e., aggregations of change groups) AggSet.

The function associateGroups(G, G’) (See Algorithm 3) returns a set of change group associations from a change group G to another change group G’, AssoTypes is a set consisting of all the association types such as Method Call.
(forward), Method Call (backward), and Similar Content (see Section III-C). For each association type type, a change group association asso is generated (Line 3-13). Its elements asso.group1, asso.group2 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 ∈ G.mem and each ch' ∈ G’.mem, the function ExistAssociation(type, ch, ch') checks whether there exists an association relationship of the type 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 3 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 function extendAggregation(agg, asso) (See Algorithm 4) takes as input a trajectory pattern agg and a change group association asso. It 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 G1 and G2 respectively. Otherwise, the algorithm returns Null. Then a new trajectory pattern newAgg is created with agg.groups and G2 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). 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 (ch1, ch2) ∈ as.insSet that satisfy either of the following two conditions:

- ch1 ∈ G.mem ∧ ch2 ∈ G’.mem ∧ (∃a ∈ assoSet, (ch, (ch1, ch) ∈ a.insSet ∨ (ch, ch1) ∈ a.insSet))
- ch1 ∈ G’.mem ∧ ch2 ∈ G.mem ∧ (∃a ∈ assoSet, (ch, (ch2, ch) ∈ a.insSet ∨ (ch, ch2) ∈ a.insSet))

The returned change operation pairs are removed from as.insSet. 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.

Algorithm 4 Aggregation Extension Algorithm

```
1: function EXTENDAGGREGATION(agg, asso)
2:   if asso.group1 ∈ agg.groups ∧ asso.group2 ∈ agg.groups then
3:     G1 = asso.group1, G2 = asso.group2
4:   else if asso.group1 ∈ agg.groups ∧ asso.group2 ∈ agg.groups then
5:     G1 = asso.group1, G2 = asso.group2
6:   else
7:     return Null
8: end if
9: newAgg = newAggregation()
10: newAgg.groups = agg.groups ∪ {G1, G2}
11: assoSet = agg.associations ∪ {asso}
12: newAgg.associations = copyAssociations(assoSet)
13: queue = [ ], doneGroups = {G1, G2}
14: queue.enQueue(G1)
15: while queue.length > 0 do
16:   G = queue.deQueue()
17:   assoSet = { }
18:   if newAgg.associations do
19:     for each as ∈ newAgg.associations do
20:       if as.group1 == G then
21:         assoSet = assoSet ∪ {as}
22:       end if
23:     end for
24:     if as.group1 == G then
25:       G' = as.group2
26:     else
27:       G' = as.group1
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
```

V. Evaluation

To evaluate whether SETGA can effectively summarize evolution trajectory from dispersed code changes, we conducted a case study with three open-source systems to investigate the following three 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 What underlying rules and problems can be revealed by the identified trajectory patterns?

A. Basic Results

In the case study, we applied SETGA to three open-source systems, i.e., jEdit\(^1\), jBPM\(^2\), Eclipse SWT\(^3\). 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. For these three systems, we analyzed their commit records from Git repositories obtained from SourceForge, GitHub, and Eclipse.org, respectively.

The following thresholds were used in the case study: \(threshold_{\text{frag}}=100\) (minimum length of a code fragment by character), \(threshold_{\text{prop}}=2\) (minimum number of common properties of a group), \(threshold_{\text{minIns}}=2\) (minimum number of instances in a group), \(threshold_{\text{maxIns}}=10\) (maximum number of instances in a group).

The basic results of the case 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 operations in each pattern (#O/T), 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 operations in each pattern (#O/T), average number of commits involved in each pattern (#C/T).

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). 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.

B. 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 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 3. 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): 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 4. In this example, a developer extracted two code fragments from two methods of the Tasklite class as separate methods, and added method calls from the original methods to the two new methods.

3) Change Method Signature (Type 3): 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 5. In this example, 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): 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 6. 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.
TABLE IV
BASIC RESULTS OF THE CASE STUDY

<table>
<thead>
<tr>
<th>System</th>
<th>Period</th>
<th>Time</th>
<th>#C</th>
<th>#F/C</th>
<th>#O</th>
<th>#G</th>
<th>#P/G</th>
<th>#O/G</th>
<th>#T</th>
<th>#G/T</th>
<th>#O/T</th>
<th>#C/T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.1-3.2</td>
<td>Apr. 2001 - Aug. 2001</td>
<td>117</td>
<td>0.98</td>
<td>3167</td>
<td>976</td>
<td>4.54</td>
<td>4.30</td>
<td>132</td>
<td>2.67</td>
<td>6.39</td>
<td>2.24</td>
</tr>
<tr>
<td></td>
<td>3.3-3.5</td>
<td>Apr. 2002 - Feb. 2003</td>
<td>329</td>
<td>3.31</td>
<td>5668</td>
<td>1511</td>
<td>3.71</td>
<td>4.03</td>
<td>235</td>
<td>2.56</td>
<td>6.26</td>
<td>2.53</td>
</tr>
<tr>
<td>jBPM</td>
<td>5.1-5.2</td>
<td>Jun. 2011 - Dec. 2011</td>
<td>324</td>
<td>1.66</td>
<td>3228</td>
<td>858</td>
<td>3.68</td>
<td>4.12</td>
<td>71</td>
<td>2.30</td>
<td>3.92</td>
<td>2.10</td>
</tr>
<tr>
<td></td>
<td>5.2-5.3</td>
<td>Dec. 2011 - May 2012</td>
<td>322</td>
<td>2.16</td>
<td>8961</td>
<td>1258</td>
<td>3.53</td>
<td>3.99</td>
<td>135</td>
<td>4.00</td>
<td>11.14</td>
<td>2.83</td>
</tr>
<tr>
<td></td>
<td>5.3-5.4</td>
<td>May 2012 - Nov. 2012</td>
<td>288</td>
<td>2.46</td>
<td>4608</td>
<td>996</td>
<td>4.02</td>
<td>4.17</td>
<td>95</td>
<td>2.82</td>
<td>6.06</td>
<td>1.87</td>
</tr>
<tr>
<td>SWT</td>
<td>3.6-3.7</td>
<td>Jun. 2010 - Jun. 2011</td>
<td>777</td>
<td>1.71</td>
<td>6113</td>
<td>1644</td>
<td>3.71</td>
<td>3.74</td>
<td>190</td>
<td>2.80</td>
<td>6.75</td>
<td>2.62</td>
</tr>
</tbody>
</table>

5) Change Inheritance Hierarchy (Type 5): 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 7. 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): A method or method call is first added/removed and then removed/added. An example from jBPM is shown in Figure 8. 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 9 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 10. 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%) can not be categorized into any basic types. 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/chaining methods or fields. There are also a large part of patterns involving repeated adding/removing the same methods or method 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 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.

C. 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 changes 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 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.

D. RQ3: 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 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 9 indicates the following convention: 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 5 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 9 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 8 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.

![Fig. 10. Distribution of Different Types of Trajectory Patterns](image-url)
VI. 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 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 sequences 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. This kind of correlations can be used for bug prediction and risk analysis.

VII. 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.

In our future work, we will further investigate the correlation between different kinds of trajectory patterns and software development quality/efficiency. Furthermore, 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.

ACKNOWLEDGMENT

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

REFERENCES