Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 650748, 9 pages
http://dx.doi.org/10.1155/2015/650748

Research Article

Composite-Level Conflict Detection in UML Model Versioning

Zu Zhang, Renwei Zhang, and Zheng Qin

School of Software Engineering, Tsinghua University, Beijing 100084, China

Correspondence should be addressed to Zheng Qin; qingzh@tsinghua.edu.cn

Received 9 February 2015; Accepted 11 May 2015

Academic Editor: Xiaoyu Song

Copyright © 2015 Zu Zhang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

At present, model-driven engineering plays an essential role in software development. Model versioning systems perform the task of conflict detection when merging parallel-developed model versions. However, conflict detection is typically conducted at the primitive operation level in operation-based systems. This situation implies that the overall intention of model developers may be disregarded, which results in unsatisfactory performance. In this study, we present an approach to conflict detection at the composite level in model versioning systems for Unified Modeling Language. This approach has two main stages. During the preprocessing stage, redundant operations are removed from the originally recorded operation lists to increase efficiency. During the conflict detection stage, a fragmentation procedure is conducted with only potentially conflicted operations allocated into the same fragment. Then, a pattern-matching strategy is applied to help indicate conflicts.

1. Introduction

Software engineering has entered an era where artifacts such as codes and documents require systematic management. In such circumstances, model-driven engineering (MDE) exerts considerable influence throughout the entire life cycle of software development. In MDE, models are regarded as first-class artifacts because they can be directly involved in code generation and serve as abstraction. As software evolves, the scale and complexity of models grow with the system, which calls for the efficient management of models.

Among these models, Unified Modeling Language (UML) models are widely employed in the industry. They are represented by several kinds of diagrams in different scenarios, including class diagrams in object-oriented software development. To ensure static model properties such as correctness and consistency, the model-checking approach [1] has been proposed to verify UML models. By adopting model modulation [2], the application range of model checking can be widened and its user-friendliness can be enhanced, particularly for individuals who have minimal experience [3].

Apart from their static properties, managing the evolvement of models is also crucial. Thus, model-specified version control systems (VCSs) are employed. VCSs that support concurrent development typically operate under an optimistic mechanism [4], which allows each developer to work on a local copy of model instances. To support collaborative editing, concurrent modified changes made by different developers have to be merged into a single stable version. During merging, conflicts among changes from parallel-edited versions may occur and have to be detected and resolved, which is a crucial task of VCSs.

The merging stage requires parallel-modified changes to be entered as inputs, which can be retrieved using either state- or operation-based approaches [5]. An operation-based approach obtains changes directly from recorded change logs; hence, the operations of arbitrary granularity can be determined as long as the editor supports it. Recorded operations are classified into primitive operations (e.g., create, delete, and update) and composite operations, which consist of primitive operations. The latter may be domain-specific such as refactorings [6]. Complex examples include Gang-of-Four design patterns [7] of UML class diagrams. These operations can provide better usability and understandability than primitive operation sequences. Furthermore, conflict detection conducted at the composite operation level may be more accurate because an integrated operation reflects the

overall intention of a developer, which is an essential criterion in declaring conflicts.

However, current operation-based versioning systems either provide limited support for detecting conflicts with composite operations during model merging [8] or conflict detection is not conducted at the composite operation level [9]; that is, composite operation is not regarded as a whole. In this study, we propose a tentative approach to detecting conflicts associated with composite operations. This approach includes a preprocessing stage that removes redundant operations [10] from recorded operations and a detection stage that identifies conflicts.

The remainder of the paper is organized as follows. In Section 2, we introduce some notions and properties employed in the study and provide a motivating example to illustrate our approach. The preprocessing and conflict-detection stages are introduced in Sections 3 and 4, respectively. Section 5 evaluates the proposed approach. Section 6 introduces related works. Finally, the conclusion drawn from the study is presented in Section 7.

2. Preliminary

In this section, several concepts, including models and operations, applied in the proposed approach are introduced. The models referred to in this study are UML models.

Definition 1. An instance of *model* M is naturally defined by the model elements E that it contains, the available operations O that can be conducted on it, and the constraints T that it complies with; that is, $M = \langle E, O, T \rangle$.

Available model elements and constraints are frequently defined by a *metamodel*, which is typically represented as a UML class diagram. A metamodel regulates the kinds of element that will be shown in the model, the relationships between elements, and the numerical and other rules applied in the model. An example metamodel is provided in Figure 1. The example is a highly simplified version of a metamodel for class diagrams (refer to [11] for the complete version). In this metamodel, class is the central element, which may aggregate several attributes or methods. A class may inherit one superclass and may generalize several subclasses. Each instance of class, attribute, and method has a name.

Definition 2. An *operation* O is specified by its target elements E, preconditions C, and postconditions C'; that is, $O = \langle E, C, C' \rangle$. Preconditions are requirements that an operation has to observe before execution, whereas postconditions are outcome characteristics.

If we denote $E \supset C_p$ when a set of elements E satisfies the preconditions of an operation p, then $p(E) \supset C'_p$ will automatically hold. In this case, C_p and C'_p are substituted for the precondition and postcondition of operation p, respectively, and $p(E)$ is substituted for the elements generated by executing p on E. This relation is expressed as follows:

$$\text{for } p \in O, \quad E \supset C_p \Longrightarrow p(E) \supset C'_p. \tag{1}$$

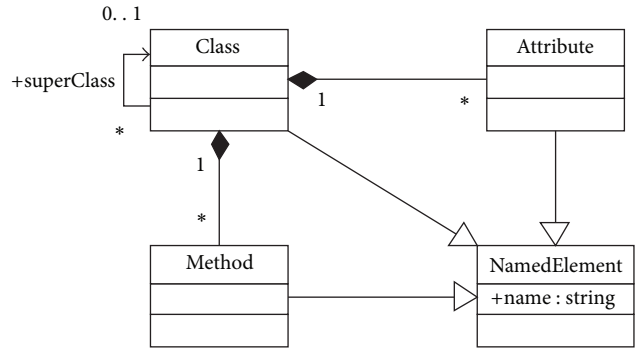


Figure 1: Simplified metamodel for class diagrams.

After an operation is defined, several crucial relations among operations are then defined as follows. Additional properties of operations are discussed in [12].

Relation 1 (equivalence). Two operations p and q are *equivalent* if they operate on the same set of elements E and share exactly the same preconditions C and postconditions C'.

Relation 2 (composite). Two operations p and q can be *composed* if $p(E) \supset C_q$ holds; that is, the preconditions of operation q are satisfied after applying p to an element set E. The composed operation is denoted as $q \circ p$, which indicates that p should be applied before q.

Relation 3 (conflict). Traditionally, an operation pair p and q are *in conflict* if either of them makes the other inapplicable or if their execution sequences lead to different results. We denote this relation as $p \between q$. Consider

$$\text{for } p, q \in O, \quad p \between q \Longleftarrow p(E) \not\supset C_q \vee q(E) \not\supset C_p \vee q \circ p \neq p \circ q. \tag{2}$$

Two operations are *compatible* with each other if they are not in conflict, which is denoted as $p \parallel q$.

However, this definition of conflict is reconsidered in our approach, particularly regarding composite operations. We may consider some operation pairs compatible if they complete similar tasks, which is introduced in Section 4.

Relation 4 (dependency). The *dependency rule* is expressed as follows, which conveys that dependency can be affirmed if the preconditions of one operation depend on the postconditions of the other. We denote $p \prec q$ if the execution of q depends on p. Consider

$$\text{for } p, q \in O, \quad p \prec q \Longleftrightarrow p(E) \supset C_q \wedge C'_p \cap C_q \neq \varnothing. \tag{3}$$

p and q are *independent* if neither $p \prec q$ nor $q \prec p$ holds.

Definition 3. Operations are classified into two sets according to whether an operation can be composed by simpler operations: *primitive operations* PO and *composite operations* CO; that is, $O = \text{PO} \cup \text{CO}$. The former often refers to an activity

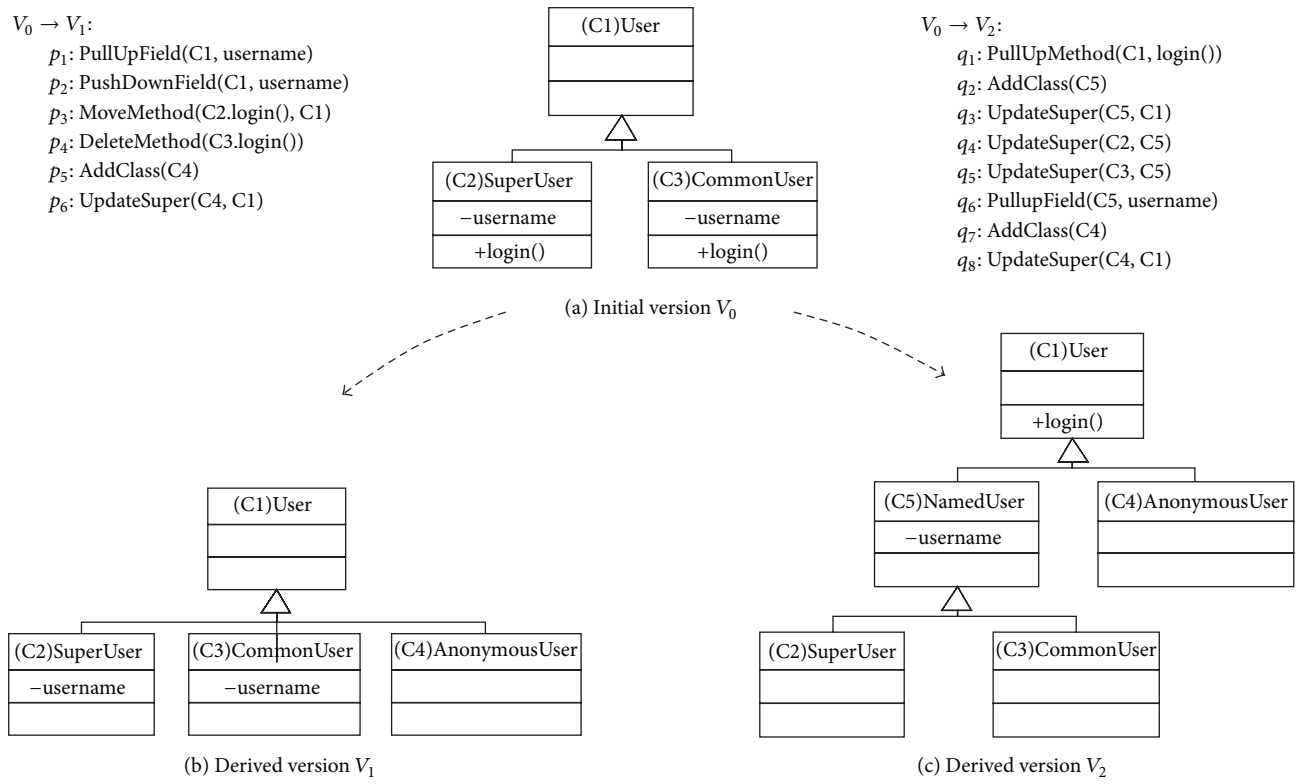


Figure 2: Motivating example.

that creates, deletes, or updates a simple element that does not contain other elements or have relationships with them. The internal representation of operations is sometimes difficult to understand for model developers [13], which is one of the reasons why composite operations should be composed of primitive ones. A composite operation can be expressed as follows:

$$\text{for } \forall x \in \text{CO},\ \exists p_i \in O,\ i = 1, 2, \ldots, n$$
$$x = p_n \circ \cdots \circ p_2 \circ p_1. \tag{4}$$

p_i can be arbitrary to whichever operation is available, regardless of whether primitive or composite. Thus, composite operations can be organized in a hierarchical manner, in which primitive operations compose low-level composite operations and low-level composite operations can further compose high-level operations such as refactorings.

An example that illustrates model changes is presented in Figure 2. A class diagram that is initially at version V_0 is edited in parallel by two developers. Thus, two derived versions, V_1 and V_2, are available after respective changes have been made by the two developers. One of the developers can commit changes directly without performing an additional process, whereas the other can commit changes only after model merging. Two sequences of recorded operations from both developers are analyzed to detect conflicts. To simplify, classes with the same label, such as C_4, from the two versions are regarded as corresponding elements with the same meaning.

The signatures of some composite operations shown in our example are briefly explained as follows and defined by the suboperation sequences in Table 1. A detailed explanation of these operations is provided in [6].

Table 1: Definition of composite operations by suboperations.

Composite operation	Sequence of suboperations
MoveField($c_1 \cdot a$, c_2)	$q \circ p$, p = AddField(c_2, a), q = DeleteField($c_1 \cdot a$)
PullUpField(c, a), c.children = $\{c_1, \ldots, c_n\}$	$p_n \circ \cdots \circ p_2 \circ p_1$, p_1 = MoveField($c_1 \cdot a$, c), p_i = DeleteField($c_i \cdot a$), $i = 2, 3, \ldots, n$
PushDownField(c, a), c.children = $\{c_1, \ldots, c_n\}$	$p_n \circ \cdots \circ p_2 \circ p_1$, p_1 = MoveField($c \cdot a$, c_1), p_i = AddField(c_i, a), $i = 2, 3, \ldots, n$

(i) PullUpField(class c, attribute a): all subclasses of class c initially have the same attribute a. Attribute a will be removed from all subclasses and created in c.

(ii) PushDownField(class c, attribute a): class c initially has attribute a. Attribute a will be removed from c and created in all of its subclasses.

Our example has two operation lists. In version V_1, the operations are recorded as p_1 to p_6. First, the common attribute *username* from C_2 and C_3 is extracted to their superclass C_1. Then *username* is pushed downward. Afterwards the method *login* is moved from C_2 to C_1 and the same method is removed from C_3. Finally a new class C_4 is added and the superclass of C_4 is set to C_1.

In version V_2, the operations are recorded as q_1 to q_8. First, the common method *login*() is extracted from C_2 and C_3 to their superclass C_1. Then, a new class C_5 is created, with its superclass set to C_1. Subsequently, the superclasses of C_2 and C_3 are set to the same class C_5 and their common attribute username is pulled upward to C_5. Finally, a new class C_4 is created and its superclass is set to C_1.

These two lists of operations both contain several composite operations, some of which are refactorings defined for class diagrams, such as pulling up a common attribute from subclasses to their shared superclass. If the operations from both sides are compared in pairs to detect conflicts, 48 pairs of operations should be checked, and complexity can be extremely high if numerous operations are included in the lists. To detect conflicts between the two lists of operations efficiently, two issues have to be resolved:

(i) Removing redundant operations: for a state-based VCS, a minimized change set can be calculated by model comparison. Meanwhile, in an operation-based VCS, redundant recorded operations are inevitable if all operations are recorded after execution.

(ii) Detecting conflicts effectively: in theory, each operation in one list should be compared with each operation in the other list to obtain a precise result. However, some comparisons are unnecessary because the compared operation may not influence one another. Therefore, analyzing only necessary pairs of operations is essential to improve efficiency.

3. Preprocessing Operations

Before conflict detection, the original recorded operation lists have to be minimized; that is, redundant operations whose effects are masked by succeeding operations should be removed. Both primitive and composite operations are considered in this process.

3.1. Removing Redundant Primitive Operations. Basic rules for omitting redundant primitive operations are provided in [14]. Primitive operations such as creating or deleting an element and updating an attribute feature can be compacted by following self-confident generic rules. In this case, e and v represent a stand-alone element and one of its property values, respectively. An empty operation that always does nothing to the model is denoted by *null*:

$$add(e) + delete(e) \rightarrow null;$$
$$add(e[v := v_1]) + update(e \cdot v, v_2) \rightarrow create(e[v := v_2]);$$
$$update(e \cdot v, v_1) + update(e \cdot v, v_2) \rightarrow update(e \cdot v, v_2);$$
$$update(e \cdot v, v_1) + delete(e) \rightarrow delete(e).$$

The preceding rules are utilized to remove redundant primitive operations in Algorithm 1. Two primitive operations that operate on the same element are compacted if no other operation that operates on the same element comes between them. Hence, the compacted operation will participate in further compaction.

```
(1)  for all element e that is operated do
(2)    operation pre = null
(3)    for all operation p in operation list do
(4)      if p does not operate on e then
(5)        continue inner iteration
(6)      else if p ∉ PO then
(7)        pre := null
(8)      else if pre = null then
(9)        pre := p
(10)     else
(11)       p := combine pre and p
(12)       pre := p
(13)     end if
(14)   end for
(15) end for
```

Algorithm 1: Removing redundant primitive operations.

In the motivating example, operations p_5/p_6 match the second rule; that is, they create an element and subsequently update its property value. Therefore, these two operations can be compacted to a single operation $AddClass(C_5[superclass := C_1])$. Operation pairs q_2/q_3 and q_7/q_8 can be similarly compacted.

3.2. Removing Redundant Composite Operations. Composite operations are typically defined in domain-specific environments; hence, rules for omitting redundant composite operations cannot be defined in a generic manner. In this case, we apply the concept of inverse operations to accomplish the task. An operation is regarded as an inverse operation of another if executing both operations on an applicable model results in no change. The inverse operation of an operation p is denoted as p^-. The following equation can be derived:

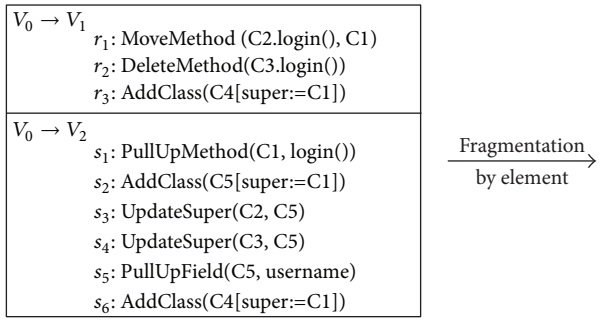
$$q = p^- \Longleftrightarrow q \circ p = null. \tag{5}$$

The inverse operation of a composite operation cp can be expressed as follows:

$$\begin{aligned} cp^- &= \left(p_n \circ p_{n-1} \circ \cdots \circ p_2 \circ p_1\right)^- \\ &= p_1^- \circ p_2^- \circ \cdots \circ p_{n-1}^- \circ p_n^-. \end{aligned} \tag{6}$$

Thus, a composite operation can be reversed by initially reversing the sequence of suboperations and then reversing each suboperation. For example, the two composite operations *PullUpField* and *PushDownField* defined in Table 1 are mutually reversed if they operate on exactly the same set of elements, which can be proven by the preceding equation. With the specified reversal rules, the mutually reversed composite operations in an operation list can be safely removed if no other operation that modifies any common element comes between them.

In our example, operations p_1 and p_2, which operate on the same set of elements, are mutually reversed composite operations based on the preceding analysis; hence, they can be removed from the operation list.



Element	Left operations	Right operations
C1.login()	r_1	s_1
C2.super		s_3
C2.username		s_5
C2.login()	r_1	s_1
C3.super		s_4
C3.username		s_5
C3.login()	r_2	s_1
C4.super	r_3	s_6
C5.super		s_2
C5.username		s_5

Figure 3: Fragment operations by elements.

4. Conflict Detection

Conflict detection is performed to check for conflicts between two operation serializations. Two issues should be addressed in conflict detection. First, operation pairs from two lists should be filtered to detect conflicts because time will be wasted if all pairs are considered [12]. Second, conflict rules, which determine whether two operations are in conflict, should be defined formally.

4.1. Fragmentation by Elements. Operation pairs to be checked should be minimized to efficiently detect conflicts between two versions. The concept of fragmentation is adopted in this study to divide operations into several partitions. Only operations from the same fragment are compared to detect conflicts. We conduct fragmentation according to the operated elements.

The fragmentation procedure is simple if the definition of operations is fine-grained. One fragment corresponds to an operated element and contains all operations that modify this element. In this case, only the simplest elements are considered, such as the attributes in a class diagram. Each fragment contains a set of operations and each operation may be contained by several elements. No conflict exists if two operations do not in any common fragment, which can be expressed as follows. $E(p)$ denotes element set related operation p. Consider

$$\text{for } p, q \in O, \quad p \parallel q \Longleftarrow E(p) \cap E(q) = \varnothing. \tag{7}$$

The fragmentation procedure is illustrated in Figure 3. The table on the left lists the operations after preprocessing, whereas the table on the right shows the fragments. The first column of the fragment table lists the involved elements. Each row represents a fragment associated with one element. Three kinds of element from each class are considered: attribute *username*; method *login*(); and relationship *super*, which refers to the superclass. During fragmentation, the two operation lists are traversed to check the elements involved in each operation. Then, each operation is placed in the rows of their related elements. After fragmentation, operations from the two lists are separately classified according to elements, which are shown in the table as *Left Operations* and *Right Operations*. In the succeeding procedure, only operation pairs (each pair contains one operation from the left and right columns) in the same row are compared to detect conflicts.

4.2. Conflict Detection with Compatibility Pattern

4.2.1. Conflicts between Primitive Operations. Conflicts between two primitive operations can only occur when both operations operate on the same element in a fine-grained model versioning system. In this case, detection can be implemented by establishing a map from the changed elements to the operations that cause the changes. A conflict can only occur if an element maps operations from both operation lists, which can be expressed as follows. In this case, e_p indicates the element operated by primitive operation p. Concrete conflict rules for primitive operations can be found in the following [15]:

$$\text{for } p, q \in \text{PO}, \quad p \nparallel q \Longrightarrow e_p = e_q. \tag{8}$$

4.2.2. Conflicts Associated with Composite Operations. Multiple elements are frequently involved in composite operations; hence, a sophisticated approach is required to accomplish precise detection.

Conflicts between a primitive operation and a composite operation occur if the primitive operation changes the same element in a different manner from that of a suboperation in the composite operation, which can be expressed as follows:

$$\begin{aligned} &\text{for } p \in \text{PO}, \ q \in O, \ cp \in \text{CO}, \\ &\qquad p \nparallel cp \Longleftarrow \exists q \in sub(cp), \quad e_p = e_q \wedge p \nparallel q. \end{aligned} \tag{9}$$

However, this is not a necessary condition; that is, other cases that result in conflicts also occur. For example, suppose a *PullUpField* operation p that requires that all subclasses have one same attribute *attr* and a delete operation q that removes *attr* of one subclass. Operation p does not conflict with any suboperation of q because q also removes *attr* from that subclass; however, conflict still occurs because the precondition of p is violated (i.e., the attribute *attr* of one subclass is missing). To detect conflicts associated with composite operations, a pattern-matching strategy is employed.

Definition 4 (equivalence pattern). Operation p, which matches the equivalence pattern of composite operation q,

Table 2: Excerpt of equivalence pattern table.

Composite operation	Equivalence pattern	
	Suboperations	Relations
PullUpField(c, a), c.children = $\{c_1, \ldots, c_n\}$	op_i: MoveField($c_i \cdot a$, c), $i \in \{1, 2, \ldots, n\}$ op_j: DeleteField($c_j \cdot a$), $j = 1, 2, \ldots, n$, $j \neq i$	—
PushDownField(c, a), c.children = $\{c_1, \ldots, c_n\}$	op_i: MoveField($c \cdot a$, c_i), $i \in \{1, 2, \ldots, n\}$ op_j: AddField(c_j, a), $j = 1, 2, \ldots, n$, $j \neq i$	—

has the same effect as q on a model instance. A composite operation may have several equivalence patterns, with each pattern containing a set of operations and their internal dependencies (c.f. Relation 4). Thus, an equivalence pattern defined by its contained suboperations *Sub* and internal dependency relations R between suboperations can be expressed as $\langle Sub, R\rangle$. We denote $P_e(p)$ as an equivalence pattern of operation p, and $P_e(p) \overset{m}{\supset} P_e(q)$ if $P_e(p)$ contains all the rules (*Sub* and R) specified by $P_e(q)$.

Equivalence patterns can be derived from the suboperation sequences in Table 1. First, the suboperation set *Sub* can be directly generalized from the definition. Then, according to the dependency rule in Section 2, dependency relations R can be obtained by checking the preconditions and postconditions of these suboperations. Some example equivalence patterns, which help navigate the logic structure of composite operations, are represented in Table 2.

Definition 5 (compatibility pattern). Operation p, which matches the compatibility pattern of a composite operation q, does not conflict with q. A composite operation may also have several compatibility patterns. A compatibility pattern can be defined in a similar manner as equivalence pattern, that is, by suboperations and their relations.

Definition 6 (minimal compatibility pattern). A compatibility pattern with a minimal set of suboperations is called a minimal compatibility pattern. Given that several compatibility patterns may exist, attempting to match each of them is a complicated task. Thus, defining a pattern that specifies the minimum set of rules, which can represent several similar patterns, is useful. The minimal compatibility pattern reflects the intention of this composite operation because it is a common set of several compatibility patterns. We denote $P_c(p)$ as the minimal compatibility pattern of operation p. The symbol $\overset{m}{\supset}$ is also applicable to compatibility patterns. For example, $P_e(q) \overset{m}{\supset} P_c(p)$ indicates that operation q matches all the rules specified by pattern $P_c(p)$.

The table of minimal compatibility patterns for our example is labeled as Table 3. Compared with equivalence patterns, compatibility patterns have fewer specified rules. For example, the suboperation *DeleteField* of operation *PullUpField* is absent from the minimal compatibility pattern and the suboperation *MoveField* that can represent the intention is reserved.

Table 3: Excerpt of minimal compatibility pattern table.

Composite operation	Minimal compatibility pattern	
	Suboperations	Relations
PullUpField(c, a), c.children = $\{c_1, \ldots, c_n\}$	op_i: MoveField($c_i \cdot a$, c), $1 \leq i \leq n$	—
PushDownField(c, a), c.children = $\{c_1, \ldots, c_n\}$	op_i: MoveField($c \cdot a$, c_i), $1 \leq i \leq n$	—

After operation patterns are specified, several rules are established to detect conflicts in the presence of composite operations.

Rule 1. Two composite operations are not in conflict if an equivalence pattern of either operation satisfies the rules of the minimal compatibility pattern of the other operation, which implies that either operation complete the main task of the other operation. Consider

$$\text{for } p, q \in \text{CO}, \quad p \parallel q \Longleftarrow P_e(p) \overset{m}{\supset} P_c(q) \wedge P_e(q) \overset{m}{\supset} P_c(p). \tag{10}$$

Rule 2. Operation p is not in conflict with composite operation q if the equivalence pattern of p satisfies the minimal compatibility pattern of q and is satisfied by the equivalence pattern of q. An operation is not in conflict with a composite operation if it accomplishes the essential part of the composite operation. Consider

$$\text{for } p \in O,\ q \in \text{CO}, \quad p \parallel q \Longleftarrow P_e(q) \overset{m}{\supset} P_e(p) \wedge P_e(p) \overset{m}{\supset} P_c(q). \tag{11}$$

Rule 3. Operation p is not conflict with composite operation q if an equivalence pattern of p is satisfied by an equivalence pattern of q and the minimal compatibility pattern of q is satisfied by another operation r. Once the minimal compatibility pattern of q has been satisfied by r, other operations that construct part of q do not conflict with q. Consider

$$\text{for } p \in O,\ q \in \text{CO},\ r \in O, \quad p \parallel q \Longleftarrow P_e(q) \overset{m}{\supset} P_e(p) \wedge P_e(r) \overset{m}{\supset} P_c(q). \tag{12}$$

The algorithm that detects conflicts by following the aforementioned rules is illustrated in Algorithm 2. *matchEP*(p, q) and *matchCP*(p, q) are used to check if

```
Inputs: Fragments: the result of fragmentation
Output: ConflictPairs: conflict operation pairs
(1) for all element e that is operated do
(2)   Init pair container ConflictPairs,
(3)     CandidatePairs, SafePairs
(4)   LeftOpSet := Fragments.getRow(e).left
(5)   RightOpSet := Fragments.getRow(e).right
(6)   for all p ∈ LeftOpSet do
(7)     for all q ∈ RightOpSet do
(8)       if matchCP(p, q) and matchCP(q, p) then
(9)         put (p, q) and (q, p) in SafePairs
(10)      else if matchEP(p, q) then
(11)        if matchCP(q, p) then
(12)          put (p, q) in SafePairs
(13)        else
(14)          put (p, q) in CandidatePairs
(15)        end if
(16)      else if matchEP(q, p) then
(17)        same code as line (11)–(15) (reverse p, q)
(18)      else
(19)        put (p, q) in ConflictPairs
(20)      end if
(21)    end for
(22)  end for
(23)  for all (r, s) ∈ CandidatePairs do
(24)    if r ∉ SafePairs.Keys then
(25)      put (r, s) in ConflictPairs
(26)    end if
(27)  end for
(28) end for
```

Algorithm 2: Detecting conflicts in operation pairs.

$P_e(p) \overset{m}{\supset} P_e(q)$ and $P_e(p) \overset{m}{\supset} P_c(q)$, respectively. Some detailed codes are dismissed for understandability. For example, a code that guarantees each operation pair is checked only once.

In this algorithm, the input is the fragmentation result, such as the right table in Figure 3, and the output *ConflictPairs* contains conflict operation pairs. Apart from *ConflictPairs*, *CandidatePairs* records pairs that are possibly in conflict, whereas *SafePairs* records those that are not in conflict. Pairs to be checked are retrieved from the left and right operations (represented by *LeftOpSet* and *RightOpSet*) in each row of the fragment table (lines (4)-(5)). In lines (8)-(9), operation pairs that conform to Rule 1 are placed under *SafePairs*. Meanwhile, operation pairs that conform to Rule 2 can be checked directly (lines (10)–(12)). Checking Rule 3 relies on the results of the other two rules, and, thus, operation pairs that potentially conform to Rule 3 are temporarily added to *CandidatePairs* (lines (13)-(14)) and are retested later (lines (23)–(27)) after checking all operation pairs from both sets.

In our example, operation pairs r_1/s_1, r_2/s_1, and r_3/s_6 are compared. r_3 and s_6 are exactly the same operation, which indicates that Rule 1 is satisfied. Operation pair r_1/s_1 complies with Rule 2 when referring to the equivalence pattern table and the minimal compatibility pattern table. Operation pair r_2/s_1 conforms to Rule 3 because r_1/s_1 has been checked with $P_c(s_1)$, which is matched by $P_e(r_1)$. Therefore, no conflict exists in these three pairs of operations.

Table 4: Test cases under different scenarios.

Operation number	PO-Left	CO-Left	PO-Right	CO-Right
Case 1	65	6	73	5
Case 2	62	37	58	40
Case 3	183	19	198	16
Case 4	192	102	178	107

5. Evaluation

In this section, we evaluate our proposed approach on several sets of input data. To illustrate the effect, we compare our approach with an existing operation-based model versioning system and provide an analysis of the results.

5.1. Test Cases. Test cases are from a subproject of a modeling system called Concept Modeling Tool (ConMoT). ConMoT aims to provide a modeling platform for developers from different companies. Several UML models are supported, such as class diagrams and entity relationship diagrams.

We compare our approach with EMFStore [9], an operation-based model versioning system that implements functions, which include model repository, operation recording, and conflict detection and resolution. We transplant our models and algorithms into EMFStore. The models in ConMoT are specified by EMF Ecore [16], which is the same technology as EMFStore, and thus they can be integrated into EMFStore. We transform several composite operations such as PullUpField into formats composed by EMFStore-specified operations, whereas primitive operations are the same as those in EMFStore. Some methods are employed to ensure that the conflict detection mechanism of EMFStore will work well on our models. For example, most of the elements are created beforehand in case newly created elements will not correspond in the two compared operation lists considering that EMFStore recognizes corresponding elements through a unique ID.

Four sets of input data are provided in Table 4. Each test case contains two operation lists of parallel-edited models. The PO-Left and CO-Left columns show the operation numbers of primitive and composite operations in one operation list, respectively, while the PO-Right and CO-Right columns show operation numbers in the compared operation list. On average, approximately 50 elements are involved in Case 1 and Case 2, while approximately 110 are involved in Case 3 and Case 4. Most operations contained in Case 1 and Case 3 are primitive operations, with a small portion of composite operations. In Case 2 and Case 4, a higher proportion of the operations are composite.

5.2. Results and Discussions. We measure our approach through precision and recall, which are two indicators derived from the information retrieval scenario. In this study, precision is equivalent to the ratio of correctly detected

Table 5: Conflict detection under different scenarios.

	Precision-ConMoT	Recall-ConMoT	Precision-EMFStore	Recall-EMFStore
Case 1	90%	89%	91%	88%
Case 2	86%	84%	72%	69%
Case 3	88%	90%	86%	88%
Case 4	83%	81%	75%	66%

conflicts to all detected conflicts. Meanwhile, recall is equivalent to the ratio of correctly detected conflicts to all actual conflicts. Given that the actual number of conflicts cannot be obtained through a perfect approach, it is estimated through the sum of correctly detected conflicts of both approaches.

The test results are shown in Table 5. In Case 1 and Case 3, in which the ratio of primitive operations is high, both approaches perform well (high ratios of approximately 90%). However, ConMoT achieves higher precision and recall than EMFStore when composite operations are more, as shown in Case 2 and Case 4 (approximately 85% versus 70%). The number of elements does not influence the results considerably, as demonstrated by comparing Case 1 with Case 3 (or Case 2 with case D).

The differences are attributed to EMFStore detecting conflict that concerned composite operations by comparing their contained primitive operations, whereas ConMoT compares them as integrated operations. EMFStore may misjudge mutually compatible composite operations as conflicts, which decreases its precision rate. Moreover, EMFStore may fail to detect conflicts when one operation destroy the preconditions of the other operations, which results in a low recall rate.

In conclusion, our proposed conflict detection approach, which regards composite operations as a whole, can perform well in operation-based versioning systems, particularly under circumstances where composite operations such as refactorings are frequently executed.

6. Related Works

In this section, we discuss related works in model versioning, particularly conflict detection techniques.

A rapid improvement in model versioning has been witnessed in recent years. Although many state-based model versioning systems have been developed [17, 18], operation-based systems have demonstrated their advantages. For example, model modifications can be immediately retrieved from change logs. CoObRA [8] is an operation-based system developed for UML models. In this system, composite operations are regarded as a set of primitive operations, and conflict detection is performed at the primitive operation level. EMFStore [9] is another operation-based modeling system that supports storage and versioning of EMF-based models. EMFStore records both primitive and composite operations and provides hooks for customized composite operations. Although this system can fully detect conflicts among primitive operations, conflicts among composite operations are detected according to whether their contained primitive operations are in conflict. The aforementioned systems do not conduct conflict detection at the composite operation level, which is the objective of our approach.

Some approaches can detect conflicts that concern composite operations. Gerth et al. [19] exploited formalized terms for business process models (BPMs), which can detect both syntactic and semantic conflicts. Given that this approach employs the structural characteristics of BPMs, for example, a maximum of one link exists between two nodes, it cannot be applied in other models used in MDE. Brosch et al. [20] detected conflicts in composite operations by matching the preconditions and postconditions of operations in a state-based environment, as well as providing precise recommendations for conflict resolution. Mens et al. [21] detected UML refactoring conflicts through graph transformation and by specifying critical pairs. Rules were specified where conflicts might occur and corresponding resolutions were provided. However, specifying all conditions where conflicts can occur between two operations may be a complex process. By contrast, our approach specifies patterns for each composite operation.

7. Conclusion

In this study, we introduce an approach for detecting conflicts among change lists in an operation-based environment. We focus on detecting conflicts at the composite operation level, which can improve the accuracy of conflict detection by considering the intention of operations. Operations are initially preprocessed to remove redundant operations and improve performance. Then, operations are classified into fragments according to their operated elements, and conflicts are located through pattern matching.

We compare our approach with an existing model versioning system, and the results show that our approach is effective. However, considerable work still has to be performed in the future, including detecting conflicts on model inconsistency and resolving conflicts according to the detection results.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] P. Zhang and G. Luo, "Research of model checking UML," *Computer Applications*, vol. 27, no. 10, pp. 2493–2500, 2007.

[2] M. Xia, G. Luo, and M. Sun, "Modeling and model checking by modular approach," in *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pp. 628–629, Hyderabad, India, May 2014.

[3] M. Xia, K. Lo, S. Shao, and M. Sun, "Formal modeling and verification for MVB," *Journal of Applied Mathematics*, vol. 2013, Article ID 470139, 12 pages, 2013.

[4] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, 2002.

[5] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Computing Surveys*, vol. 30, no. 2, pp. 232–282, 1998.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, X-Temp, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: abstraction and reuse of object-oriented design," in *ECOOP '93—Object-Oriented Programming*, vol. 707 of *Lecture Notes in Computer Science*, pp. 406–431, Springer, 1993.

[8] C. Schneider, "CoObRA—a small step for development tools to collaborative environments," in *Proceedings of the 26th International Conference on Software Engineering (W2S Workshop '04) 'Workshop on Directions in Software Engineering Environments (WoDiSEE '04)'*, vol. 2004, pp. 21–28, 2004.

[9] M. Koegel and J. Helming, "EMFStore—a model repository for EMF models," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, vol. 2, pp. 307–308, May 2010.

[10] E. Lippe and N. van Oosterom, "Operation-based textbased merge tools," in *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments*, pp. 78–87, ACM, Washington, DC, USA, December 1992.

[11] OMG, *UML Infrastructure Specification, v2.4.1*, OMG, 2011.

[12] M. Koegel, M. Herrmannsdoerfer, O. Von Wesendonk, and J. Helming, "Operation-based conflict detection," in *Proceedings of the 1st International Workshop on Model Comparison in Practice (IWMCP '10)*, pp. 21–30, July 2010.

[13] T. Kehrer, U. Kelter, and G. Taentzer, "A rule-based approach to the semantic lifting of model differences in the context of model versioning," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, pp. 163–172, IEEE, Lawrence, Kan, USA, November 2011.

[14] A. Rajbhoj and S. Reddy, "A graph-pattern based approach for meta-model specific conflict detection in a general-purpose model versioning system," in *Model-Driven Engineering Languages and Systems*, vol. 8107 of *Lecture Notes in Computer Science*, pp. 422–435, Springer, Berlin, Germany, 2013.

[15] K. Altmanninger, M. Seidl, and M. Wimmer, "A survey on model versioning approaches," *International Journal of Web Information Systems*, vol. 5, no. 3, Article ID 1806332, pp. 271–304, 2009.

[16] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, Addison-Wesley Professional, 2008.

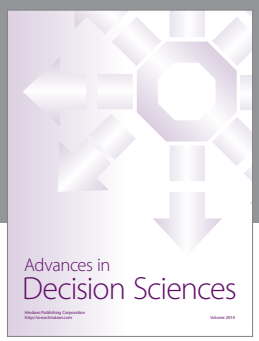
[17] K. Altmanninger, A. Bergmayr, and W. Schwinger, "Semantically enhanced conflict detection between model versions in SMoVer by example," in *Proceedings of the Models in Software Engineering, Workshops and Symposia (MoDELS '07)*, pp. 293–304, 2007.

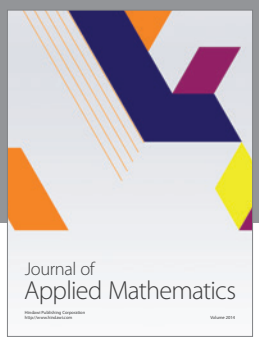
[18] K. Altmanninger, G. Kappel, A. Kusel et al., "AMOR towards adaptable model versioning," in *Proceedings of the 1st International Workshop on Model Co-evolution and Consistency Management in conjunction with MODELS 08*, pp. 55–60, 2008.

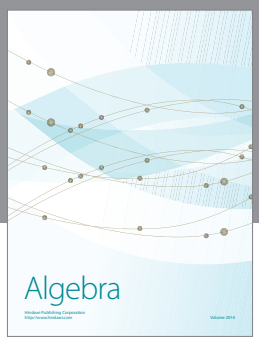
[19] C. Gerth, J. M. Küster, M. Luckey, and G. Engels, "Detection and resolution of conflicting change operations in version management of process models," *Software and Systems Modeling*, vol. 12, no. 3, pp. 517–535, 2013.

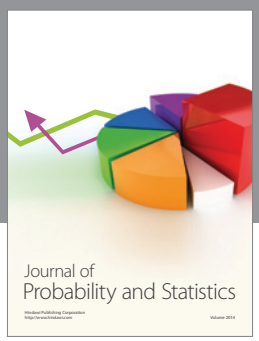
[20] P. Brosch, G. Kappel, M. Seidl et al., "Adaptable model versioning in action," in *Proceedings of the Modellierung (Modellierung '10)*, vol. 161, pp. 221–236, March 2010.

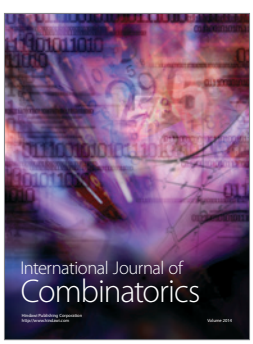
[21] T. Mens, G. Taentzer, and O. Runge, "Detecting structural refactoring conflicts using critical pair analysis," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 3, pp. 113–128, 2005.

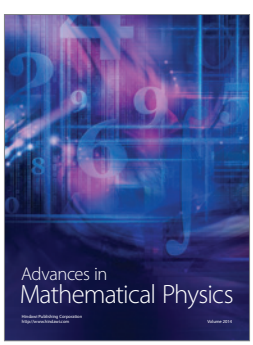


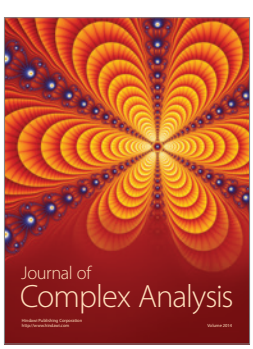


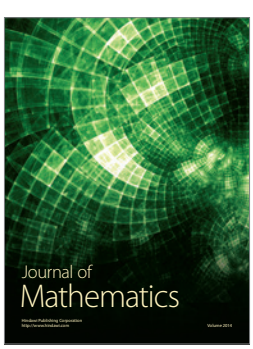


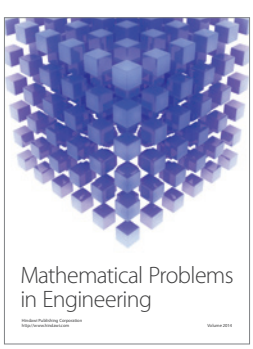


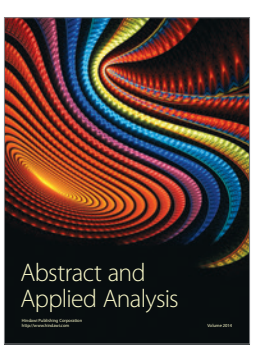


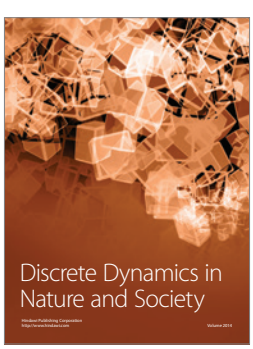


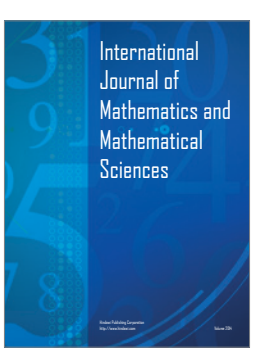












Hindawi

Submit your manuscripts at
http://www.hindawi.com