IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 15, 2016, accepted December 30, 2016, date of publication January 9, 2017, date of current version March 6, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2649565

Detection of Composite Operation in Model Management

RENWEI ZHANG[1], ZHENG QIN[1], HOUBING SONG[2], SHENGNAN LI[1], AND XIAO YANG[1]

[1]School of Software, Tsinghua University, Beijing 100084, China
[2]Institute of Technology, West Virginia University, Morgantown, WV 26506, USA

Corresponding author: R. Zhang (zrw12@mails.tsinghua.edu.cn)

ABSTRACT Model management systems become increasingly critical in model-driven engineering. One of the main tasks of these systems is to record the operations performed on model elements. While most systems support the record of primitive model change operations, complex composite model change operations are neglected, which may result in the lack of understandability. In this paper, we propose an approach to capture model transformation from primitive operations to composite ones. First, based on the low-level operation, we define some general high-level operations with hierarchical structures. Then, a matching algorithm is designed to compare primitive operations with the hierarchical structures from the bottom up. If matching successfully, the primitive operations would be lifted to a composited operation. The algorithm is iterative and ensures that all operations are lifted. The evaluation results on real-world cases show that both precision and recall of composite operation detection are improved when compared with the EMF Modeling Operations (EMO) and Complex Change Detection Engine (CCDE) algorithms.

INDEX TERMS Model management, composite operation with hierarchical structures, operation detection.

I. INTRODUCTION

Nowadays, model-driven engineering (MDE) plays an essential role in system design [1]–[3]. Models are the central artifact throughout the whole life-cycle of system development in MDE [4], [5]. They are not only an profile of the system, but also raw materials for verification, validation and code generation [6]–[10]. As the system models may evolve now and then, model management, such as model versioning control system (MVCS) [11] are consequently employed. One of the most challenging task in these systems is how to track changes of model elements effectively, such as creation and deletion.

Approaches for change tracking are mainly classified into state-based ones and operation-based one [12]. State-based approach extracts changes from model itself with a model differencing phase [13], while operation-based approach is corresponding to the performed commands [14]. However, most existing mainstream MVCS [15]–[18], whether state-based or operation-based, record only primitive operations which are recognized as the simplest operations that create, delete or modify one model element. On the other hand, as containing domain-specific information of relative models, composite operation [19] which are composed by lower level operations with a common purpose often contributes to better understandability and higher performance of related activities like conflicts detection [20] and model transformation [21], which is obviously more useful in practice. Therefore, the detection of composite operation becomes of the utmost importance.

Currently, most works on composite operation detection [22]–[25] usually cannot reach high precision or recall for several reasons. Firstly, for a given operation sequence different composite operations could be composed, as there might be overlapping part in different composite operations and we cannot choose the right one without extra information such as user's intention. Moreover, the length of operations in a composite operation might be uncertain since there might be more than one operations for the same operation type, thus the sub-operations would often be incomplete when detecting a composite operation, a.k.a ***Indefinite length problem*** mentioned in [25]. Finally, while validating the preconditions of a composite operation, different order of primitive operations might produce different results [22], which would also influences the detection accuracy. Besides, the search space of potential composite operation list might be very large in some model languages [23], and in such cases it can be very time-consuming that enumerates every possible solutions.

To counteract these problems, we introduce a novel approach based on a multi-level hierarchical definition of composite operation and a level-lifting detecting algorithm,

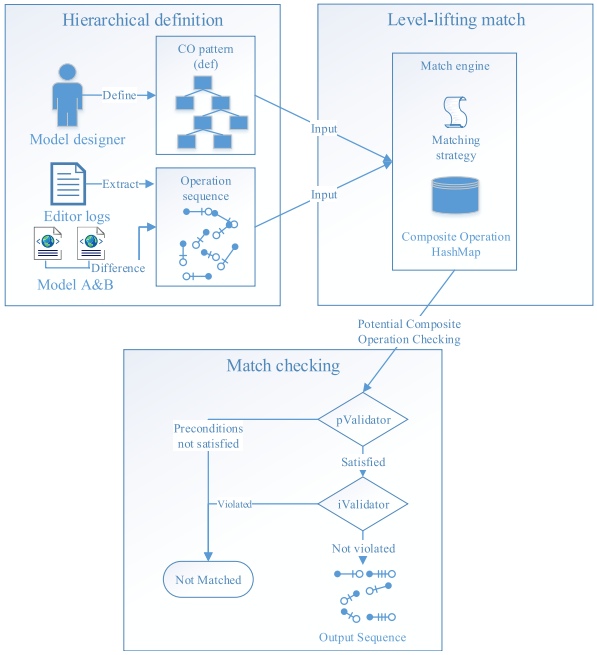


FIGURE 1. Workflow of the detection procedure.

which can be used in both state-based and operation-based environments. The overview of the approach is described in Fig. 1, and details would be illustrated with the operations on UML class diagrams throughout the paper. In the hierarchical definition step, both primitive and composite operations required in class diagrams as well as relative relations are defined. Each composite operation is first defined directly by primitive ones and then reorganized in a hierarchical structure. In the level-lifting step, a matching procedure is conducted. Before matching progress start, obtaining primitive operation sequence and the dependency relations between low-level operations would be taken into account first. Then an iterative level-lifting algorithm is employed, which contains a *top-down* scanning and a *bottom-up* compositing phase. During each iteration low-level operations attempt to match one certain composite operation. At last, invalid composite operation will be discharged in the match checking stage.

The main contributions of this paper are as follows:

- We propose a novel composite operation definition with hierarchical structure based on *pattern* containing all the sub-operations' type with multiplicity and relation constraint, which helps to avoid the ***Indefinite length problem*** and makes model developers to detect the complex composite operations with great clarity
- We propose an effective level-lifting detecting algorithm based on this hierarchical structure, which can reuse the existing matching results and achieve a higher performance.
- We design a new checking system to validate not only the satisfiability of preconditions of each composite operation even in the condition that the operations sequence is unordered, but also whether the irrelevant operations in operation sequence violate the composite operations.

The rest of the paper is organized as follows. In Section II initial definitions and relations of primitive operations and composite operation are presented. Then the matching procedure is illustrated in Section III and we will evaluate our approach in Section IV. Some related works are introduced in Section V and then Section VI concludes this paper.

II. FORMAL DEFINITION

Before reviewing the primitive and composite operations in Class Diagrams of UML model, we will firstly give a refactoring example. Figure 2 shows an example of UML refactoring, in which classes named *Professor* and *Student* composed the original model. Considering their common field *name*, in the evolved model the developer created a parent class *person*, and then the field *name* in both children classes were pulled up in the new class. The operation-based record and state-based record of this change procedure are listed in the left side and right side of the figure respectively.

Follow the definition of [26], several concepts and relations, such as models and operations will be introduced, with the above refactoring example for detail illustration.

Definition 1: An instance of model $\boldsymbol{M}$ is defined by model elements $\boldsymbol{E}$ that it contains, the available operations that can be conducted on it, and the constraints $\boldsymbol{T}$ that it complies with; that is, $M = \langle E, O, T \rangle$ [26].

Take the evolved model in Fig. 2 for example. Both Class *Professor* and Field *Name* are model elements, and *AddClass(person)* is an available operation which was conducted on the model instance. Constraints T are frequently defined by a metamodel, which 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. For instance, the metamodel of class diagrams states that 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 so on.

Definition 2: An ***operation O*** is specified by its target elements $\boldsymbol{E}$, preconditions ***preC*** and postconditions ***postC***, i.e. $O = \langle E, preC, postC \rangle$. Preconditions are the prerequisite for operation executing, whereas postconditions are those declaration of the properties which have to be held after execution.

In the operation *AddField(name, person)*, for example, Class *person* and Field *name* are the target elements, and the precondition person $\rightarrow$ notEmpty() means that Class *person* should be existence before the operation executed while postcondition *name* $\rightarrow$ *notEmpty()* means that Field *name* would be created in Class *person* after execution. We also define the applicability of the operations and some other relations among the operations as follows.

Definition 3 (Applicability): An operation p can be applied to model instance M, if M satisfies the preconditions

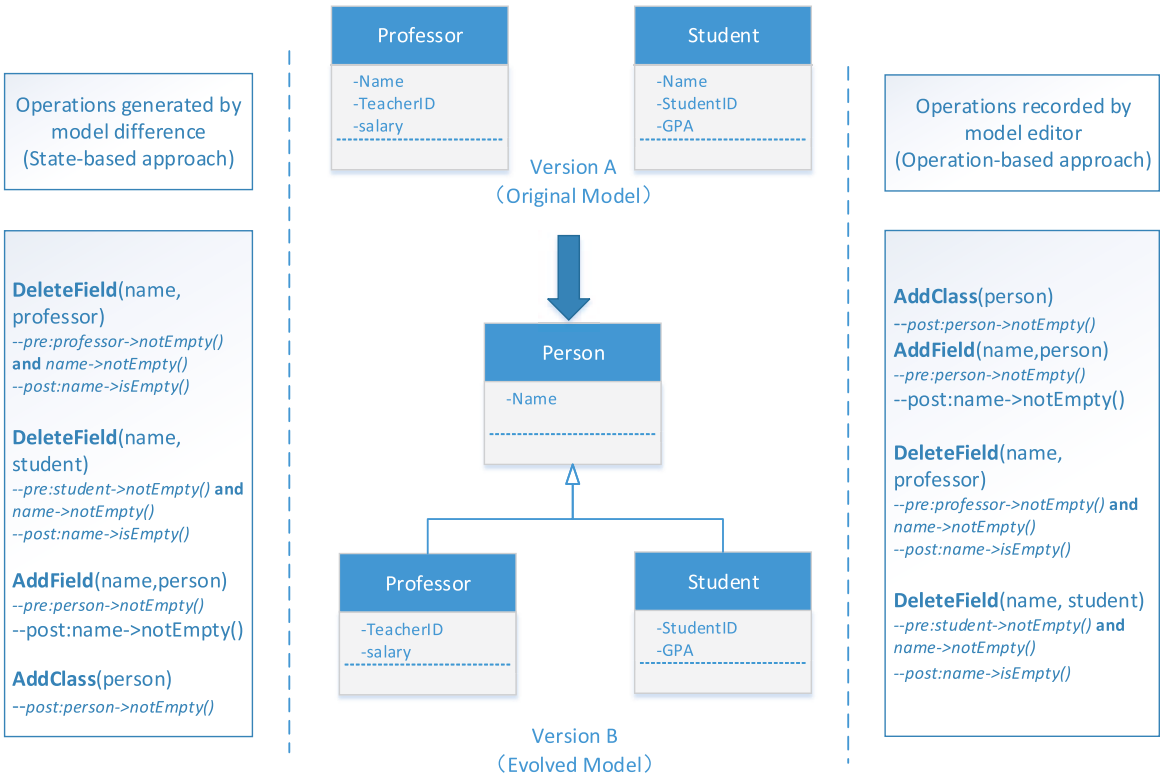


FIGURE 2. A refactoring example.

and postconditions of p, i.e.

$$p \xrightarrow{A} M \Leftrightarrow p.preC(M) == true \&\& p.postC(M) == true. \tag{1}$$

Here, we denote $p \xrightarrow{A} M$ to indicate that p is applicable to M while $p \xrightarrow{N/A} M$ means not applicable. P.preC(M) and p.post(M) represent the preconditions and postconditions when operation p would be applied to M, respectively.

As shown in Fig. 2, *AddField(name, person)* can be applied to the evolved model because in the evolved model both Class *person* and Field *name* are not empty, which just satisfies the precondition and postcondition. In the orignal model, however, Class *person* doesn't exist so that *AddField(name, professor)* is not applicable to the orginal model.

Relation 1 (Composability): Two operations p and q can be *composed* if $q \circ p \xrightarrow{A} M$. Here $q \circ p$ represents the composed operation, which indicates that p is applied before q.

In the example above, *AddClass(person)* and *AddField (name, professor)* in the right operation list can be composed because after executing *AddClass(person)* the Class *person* would not be empty, which satisfies the precondition and postcondition of the composed operation, that is, the composed operation can be applied to the original model.

Relation 2 (Dependency): An operation p is depended on by another operation q, if the execution of q requires p is executed first, which is denoted by $p \prec q$. The following rule expresses that operation q depends on operation p if q is not applicable to model M before p is applied.

$$p \prec q \Leftrightarrow q \xrightarrow{N/A} M \wedge q \circ p \xrightarrow{A} M. \tag{2}$$

Relation 3 (Violation): An operation p violates another operation q, if the execution of p makes q not applicable, which is denoted by $p \not\prec q$. The violation can be judged as:

$$p \not\prec q \Leftrightarrow q \xrightarrow{A} M \wedge q \circ p \xrightarrow{N/A} M. \tag{3}$$

In the orginal model, *AddField(name, professor)* is depended on by *AddClass(person)* since *AddField* cannot be applicable without *AddClass*, i.e. $AddClass(person) \prec AddField(name, person)$. On the other hand, if an operation *DeleteClass(person)* is inserted after *AddClass(person)* in the case above, *AddField(name, person)* would be not applicable, i.e. $DeleteClass(person) \not\prec AddField(name,person)$. Operations can be usually classified into two sets according to whether the operation can still be decomposed: *primitive operations* and *composite operations*, which will be expatiated in what follows.

A. PRIMITIVE OPERATIONS

Primitive operations accomplish the atomic modifications directly detected by the modeling tools or computed by model difference tools, which cannot be decomposed to simpler operations. These modifications are usually conducted the basic elements of a specific model. According to the category of these modifications, We denote ***primitive operation type***

by a set of primitive operations that can be applied to a model instance [27], which is defined by two parts:

- Types of atomic modification - add, delete, and update.
- Types of target model elements, i.e. classes, relationships, fields, operations.

Primitive operation type concerned in Class Diagram are listed in Table 1. Operations on method are not listed because they are similar to those on fields.

TABLE 1. Primitive operation types in class diagram.

Primitive operation type	Description
AddClass(Class c)	add class c to model
SetSuper(Class c, Class p)	set super class of c to p
DeleteClass(Class c)	remove class c from model
UpdateClass(Class c_1, Classc_2)	update classc_1 to c_2
AddField(Field e, Class c)	add field e to class c
DeleteField(Field e, Class c)	remove field e from class c
UpdateField(Class c. Field e_1, Field e_2)	update field e_1 of class c to e_2

Therefore, *primitive operations* can be defined as operations that conform the primitive operation type. For example, *AddField(Field e, Class c)* is a primitive operation type which means that add a field to a given class. Meanwhile, *addField(name, person)* which means add *name* to *person*, is the specific primitive operation. Moreover, it is important to note that the definitions and relations of operation listed above are still suitable for the primitive operation.

B. COMPOSITE OPERATIONS

A *composite operation* is composed with lower level operations, which means that it takes the exactly same effect on model elements as its composed operations. Current researches often define a composite operation as a sequence of primitive operations. However, there would be some insufficient that kept this definition from being perfect as has been argued in section I. In this paper, we define composite operation in a hierarchical structure based on composite operation pattern, which would be expounded in detail as follows.

Like *primitive operation type* defined in primitive operation, we denote *composite operation type* or *pattern* by a set of composite operations that can be applied to a model instance, which is defined by:

- A set of **operation types** which composed in a pattern. The type could be either primitive operation type or composite operation type.
- **Multiplicity constraint** of each type, i.e. the range of operations in each operation type [27].
- **Relation constraint** between its contained types [27]. For example, name of Field e in both Class c1 and c2 should be the same, when defined the pattern MoveField(Class c1.Field e, Class c2) which is composed of AddClass(Class c2,Field e) and DeleteClass(Class c1, Field e).

Therefore, a composite operation can be defined as a sequence of operations, which 1) comprise all the types composed in its associated pattern, 2) satisfy the multiplicity and relation constraint of each type, and 3) the preconditons and postconditions of each contained operation should also be satisfied.

Considering the fact that a pattern can also comprise other pattern, so it is a recursive definition of a composite operation. We also introduce mid-level operations as inter-mediate between primitive and complex composite operations. In the presence of intermediate operations, a complex composite operation can be indicated by a hierarchical structure, i.e. complex operations can be decomposed into mid-level ones and mid-level operations can be further decomposed to primitive ones. In UML Class Diagrams we define several customized mid-level composite operation types below.

- MoveField(Class $c1$.Field e, Class $c2$): move field e from class $c1$ to $c2$.
- MoveMethod(Class $c1$.Method m, Class $c2$): move method m from class $c1$ to $c2$.

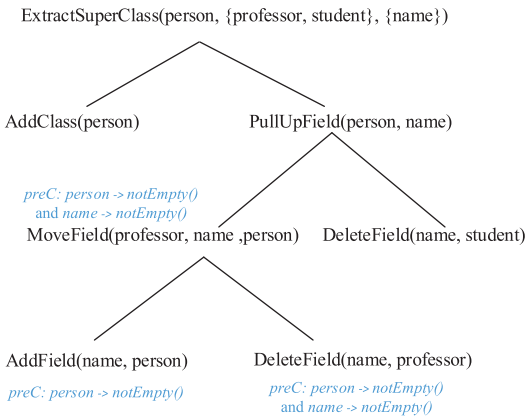


FIGURE 3. Hierarchical structure of ExtractSuperClass.

According to the patterns described in Table 2, for example, we can construct the tree structure of the ExtractSuperClass(person, professor, student, name) as in Fig. 3, where the root is the operation that is decomposed. Two classes *professor* and *student* are extracted here, which have the common field *name*, and a new class *Parent* is created as a superclass after extraction. In a first decomposition, the initial operation is split into two operations: *AddClass(person)* creates the superclass *person* and *PullUpField(person, name)* extracts the field *name* from *professor* and *student* to *person*. The *PullUpField(person, name)* is further decomposed into *MoveField(professor, name, person)* which moves field *name* from *professor* to *person*, and *DeleteField(name, student)* which removes *name* from *student*. Finally, the MoveField(professor, name, person) composes *AddField(name, person)* that adds *name* to *Parent* and *DeleteField(name, professor)* that deletes *name* from *professor*.

Moreover, the Dependency and Violation are also extended as,

Relation 4 (Dependency of Composite Operation): An composite operation p depends on another operation q, if one of its contained operation r depends on q, which can be

TABLE 2. Patterns of the rafactoring example.

Pattern	type	Multiplicity	Relation
ExtractSuperClass(Class p, ClassSet $children$, FieldSet $fields$)	addClass(Class p) PullUpField(Class $parent$, Field e)	1 1	$p.name == parent.name$
PullUpField(Class p, Field e)	MoveField(Class c_1, Field e, Class c_2) DeleteField(Field e, Class c)	1 $[1, *]$	$e.name == e.name$
MoveField(Class c_1, Field e, Class c_2)	AddField(Field e, Class c_1) DeleteField(Field e, Class c_2)	1 1	e.name==e.name and $c_2.name! = c1.name$

expressed as follows:

$$q \prec p \Leftarrow q \prec r \wedge r \in O(p). \tag{4}$$

Relation 5 (Violation of Composite Operation): An operation q violates a composite operation p, if q violates one of the contained operation of p.

$$q \not\prec p \Leftarrow q \not\prec r \wedge r \in O(p). \tag{5}$$

$O(p)$ represents all the operations composed in p.

III. DETECTION OF COMPOSITE OPERATION

In the previous section, the hierarchical structure of operations are defined, which will help detect the composite operations. The detecting procedure contains 3 stages including hierarchical definition, level-lifting match and match checking, which is illustrated in Fig. 1.

A. HIERARCHICAL DEFINITION

Two kinds of input data have to be prepared in this period: the sequence of primitive operations which are recorded by the modeler editor or computed by model difference tools, and the refactoring patterns of composite operations which are designed by the model developer according to the user requirements as well as dependency between operations.

B. LEVEL-LIFTING MATCH

In this stage, each composite operation defined in the model is tried to be matched by this sequence of operations, while each operation in the sequence will be compared to predefined patterns in a level-lifting manner iteratively. The level-lifting algorithm contains two main phases: a *top-down* scanning which searches for existing composite operations recorded in the hashmap from higher lever to lower lever and a *bottom-up* compositing which would attempt to match the lower-level operations to a higher level operations iteratively.

1) TOP-DOWN SCANNING PHASE

The input data recorded by modeler editor are all primitive operations which could be contained in any composite operation patterns. On the other hand, composite operation patterns can overlap each other, and sometimes one pattern can be even fully contained into another pattern. For example, as shown in Fig. 3, to extract a common Field *name* from Class *professor* and *student*(ExtractSuperClass) contains the process of move the Field *name* from *professor* to *person*(MoveField). Therefore, it will improve the matching efficiency dramatically by discharging the patterns whose contained patterns have not been detected before. That is, the key point of the ''top-down'' phase is to identify all the contained patterns in each composite operation pattern based on reusing the existing match results.

For an complex composite operation with a multi-level structure, each level should have more than one composite operation except the lowest level where the operations are all primitive operations. When the scanning begins, the algorithm first checks whether current level's composite operations *cco* can be found in the hashmap *comap* from the root of *patt*, and it will scan the next level if *comap* doesn't contain *cco*. This process would not be end until current level is in the leaf nodes of *patt*, or current level's composite operations are all matched from *comap*. Once scanning phase ends, corresponding operations in the sequence will be replaced by the *cco*, and then the ''bottom-up'' compositing phase begins.

2) BOTTOM-UP COMPOSITING PHASE

The bottom-up compositing process is illustrated in Algorithm 1. There are two sets of input data: *inputlist* is the operation list that to be lifted and *patt* represents the pattern that expects to detect. Considering the results of previous phase, different solutions are offered here. If composite operations *cco* in targetLevel's pattern were all found in *comap*, each operation *o* in *inputlist* would be firstly checked by sequence(line 5) and then processed in the following code according to the checking result, i.e. whether the operation *o* belongs to composite operations in targetLevel's pattern. When the composite operations are all matched, the algorithm will continue to search for the other primitive operations near the composite operations in the sequence. In another case all the operations in targetLevel's pattern belong to primitive operation, so the algorithm would search for the primitive operations in the sequence directly. If it can be managed to find all the operations in targetLevel's pattern and the multiplicity and relation constraints can be satisfied at the meantime, a higher composite operation would be substituted for the associated operations in *inputlist* and sent to the *comap*. Then the targetLevel would be lifted and a repetitive process would be continuing until the operations cannot compose a higher level operation in the end, and operations in *inputList* will be moved to *outputlist* finally.

Depending on the matching result, each operation p in the sequence could be classified into one of the following types.

- Potential Matched Operation. Operation p in the sequence might match some operation in *def*, i.e. $p \in O(def)$.
- Irrelevant Operation. That is, operations not matched.

Algorithm 1: Hierarchy Level Lifting Algorithm

Input: *patt* : pattern expected to be detected; *inputList* : initial operation list; *cco* : composite operations detected in top-down phase; *comap* : hashmap of detected composite operations;

Output: *outputList* : operation list after level lifting;

```
1: if cco matched all composite operations in pattern of targetLevel then
2:    tempOperations := comap.get(cco);
3:    replace tempOperations in inputlist with cco;
4:    clear cco
5:    for all o in inputList do
6:       if o does not match any composite operation in pattern of targetLevel then
7:          continue inner for;
8:       end if
9:       init list of candidate composite operations cco;
10:      add o to cco;
11:      if all composite operations of targetLevel are matched then
12:         continue searching for other primitive operations;
13:      else
14:         continue inner for;
15:      end if
16:      search other primitive operations others near the composite operations in the sequence;
17:   end for
18: else
19:   for all o in inputList do
20:      if o does not match any operations in pattern of targetLevel then
21:         continue inner for;
22:      end if
23:      init list of candidate operations cpo;
24:      add o to cpo;
25:   end for
26: end if
27: if all operations of targetLevel are matched && multiplicity and relation constraints are satisfied then
28:   obtain target operation instance targetOperation;
29:   replace relative operations in inputList with targetOperation;
30:   put targetOperation into comap;
31: end if
32: if targetOperation not matched then
33:   Return "notmatched";
34: else
35:   if targetLevel is root of patt then
36:      Return "matched";
37:   else
38:      Lift targetLevel;
39:      Repeat process "forall";
40:   end if
41: end if
```

Algorithm 2: Post Order Preconditions Checking Algorithm

```
1: PreconditionsChecking( Operation co) {
2: for all suboperation in co.suboperations do
3:    PreconditionsChecking(suboperation);
4: end for
5: if suboperation.preC() == TRUE then
6:    executes suboperation;
7: else
8:    return false;
9: end if
10: if suboperation.postC() != TRUE then
11:    return false;
12: end if
13: if co is root then
14:    return true;
15: end if
16: }
```

C. MATCH CHECKING

The potential matched operations detected in Algorithm1 could not be composed when the preconditions of these operations are not satisfied, and almost all the current approaches have paid enough attention to validate the preconditions. However, there will be still conflict between potential composite operations and some irrelevant operations. For example, for a given operation sequence *addField(c,a)*, *deleteField(b,a)*, *deleteClass(c)*, primitive operations *addField(c,a)*, *deleteField(b,a)* might be composed as *MoveField(b,a,c)* if preconditions are satisfied; nonetheless, irrelevant operation *deleteClass(c)* in the sequence would violate the *MoveField(b,a,c)* and then influence the detection accuracy.

In this paper, we design a new checking system to solve problems above, which include two validators: *pValidator* to check the satisfiability of preconditions, and *iValidator* to check whether the irrelevant operations in operation sequence violate the composite operations. A post-order traveler based checking algorithm performed in *pValidator* as in Algorithm2. The algorithm first travels the operations in composite operation *co*. If the preconditions of current operation are satisfied, this operation will be executed and the environment of model would be changed correspondingly. Otherwise the preconditions and postconditions cannot be fulfilled and the algorithm returns false.

On the other hand, *iValidator* will validate all the sequence of unmatched operations again and identify the Violated Operations from the irrelevant operation set. Here, Violated Operation can be defined as follows.

- Violated Operation. Operation p in the sequence violates def, i.e. $p \not\prec def$.

After the match checking stage, the matched primitive operations in the sequence would be replaced by correspond-

ing composite operations. The detection algorithm based on the multi-level structure definition offers a number of distinct advantages. First of all, the detection of composite operations can achieve a higher performance. For one thing, considering the multi-level hierarchical structure, the existing matching result in the lower level can be reused when detecting the higher level composite operations with the help of a hash map which maintain lower level composite operation matching result. For another thing, patterns whose contained composite operation are not included in the operation sequence could be discharged in the top-down scanning phase, which would reduce search space in the Bottom-up compositing phase.

Secondly, by setting the multiplicity and relation constraints of each operation type explicitly in a composite operation pattern, all possible operations in a composite operation would be found during the detecting phase, which could solve the *Indefinite length problem* accordingly.

Thirdly, overlapping parts in different composite operations often exist in lower mid-level forms(e.g. *MoveField()*), so we can improve the recall if all the composite operations based on these lower mid-level operations recorded previously could be output. In addition, according to the heuristic method *h1* mentioned in [25], the higher level operation often gets a higher level priority, thus keeping the higher level composite operations contributes to improve the precision of the detection. Next section will present the evaluation of our method.

IV. EVALUATION

In this section, we will perform our approach on real-world models to evaluate the *precision* and *recall* of the detection results.

A. EXPERIMENT SETUP

To evaluate our approach, models and operations are required. We employ the model versioning tool EMFStore [17] to accomplish the model building and operation recording. EMFStore is an EMF-based tool which supports developing models that conform to specified metamodels. When the models are modified, the operations, both primitive and composite are recorded in change logs by built-in operation recorder. The primitive operations are well-defined in EMFStore, but the composite operations have to be customized since EMFStore is designed for general purpose. We implemented Hierarchical Level-lifting Algorithm(HLLA) based on EMFStore to detect composite operations that are applicable for class diagrams in UML models, as well as mid-level operations mentioned in Section II. Moreover, we also implemented the matching algorithms mentioned in [24] and [25] for comparison.

B. INPUT DATA

The purpose of evaluation is twofold. First, whether our approach can correctly lift low-level operations to high-level ones and whether all operations that can be composed are lifted. Second, how this approach can benefit the modeling tools, e.g. compress the operation list that is recorded.

To achieve the goals above, two sets of input data named case1 and case2 are prepared. Both of them are derived from the UML Class Diagram of a real-world system called *DMPlatform* which has evolved more than 300 versions since 2013. More than 10 developers of our group have participated in the project. Operations in case1 are designed manully to cover all the composite operation patterns. We have added some exceptional case in the operation sequence to test whether the three algorithms can detect all the composite operation properly. On the other hand, in case2 we extracted primitive operations from two versions of Class Diagram as input data. The first version was a ''middle version'' when we first implemented the basic function of the system, while in the second version we have made a lot of changes to refactor the system, including 106 composite operations.

C. MEASUREMENT

In this experiment, two indicators *precision* and *recall* are computed to assess the accuracy of all three algorithms. In the scenario of operation level lifting, precision is the ratio of the correctly lifted composite operations to all lifted operations, i.e. the actually correctly lifted operations. Recall denotes the fraction of correctly lifted operations among operations that should be lifted, i.e. operations that do not fail to be lifted.

The matching results of case studies are illustrated in Table 3. The first column listed the most common composite operations in code refactoring, while the next columns showed all the detailed data to compare precision and recall of the three algorithms. It can be obviously observed that for some composite operations which accomplish a few modifications such as MoveField, RenameField, etc, the matching results of our approach comes near to perfection. For higher level operations such as Flatten Hierarchy and Extract SubClassFiled, the number of wrongly lifted operations rises and so does that of operations missed to be lifted. This is because a sequence of operations may be interpreted in different ways, e.g. when it is legal to combine one operation with either its predecessor or successor.

EMO and CCDE are composite operation detecting algorithms mentioned in [24] and [25], respectively. For EMO, the overlapping sequences of composite operation patterns would influence the recall value [24]. As shown in Case1, the composite operation Flatten Hierarchy has not been detected because the relative primitive operations are lifted as Push Down Field. On the other hand, two issues should be care about in the CCDE method. First, redundant composite operations would be generated when overlapping sequences existed in different composite operations, which would cause a quite low precision value if without extra work. Second, the order of operations seems to have an affection on the recall value, e.g. one PullUpField operation failed to be composed only because another delete AttributeOperation executed early. Finally, both EMO and CCDE ignore the validation of conflicts between potential composite operations and irrele-

TABLE 3. Match result of case1.

Composite Operation		Expected			Correct			Wrong			Precision			Recall		
		emo	ccde	hlla	emo	ccde	hlla	emo	ccde	hlla	emo	ccde	hlla	emo	ccde	hlla
Move Field	Case1	20	20	20	20	20	20	1	79	0	0.95	0.20	1.00	1.00	1.00	1.00
	Case2	13	13	13	13	13	13	1	30	0	0.92	0.30	1.00	1.00	1.00	1.00
Move Method	Case1	20	20	20	20	20	20	3	61	0	0.86	0.24	**1.00**	1.00	1.00	1.00
	Case2	9	9	9	9	9	9	0	12	0	1.00	0.45	1.00	1.00	1.00	1.00
Rename Class	Case1	20	20	20	20	20	20	3	3	0	0.86	0.86	**1.00**	1.00	1.00	1.00
	Case2	6	6	6	6	6	6	0	0	0	1.00	1.00	1.00	1.00	1.00	1.00
Rename Field	Case1	20	20	20	17	20	20	2	2	0	0.89	0.90	1.00	0.85	1.00	1.00
	Case2	31	31	31	30	31	31	3	3	0	0.90	0.91	1.00	0.97	1.00	1.00
Rename Method	Case1	20	20	20	20	20	20	2	2	0	0.90	0.90	1.00	1.00	1.00	1.00
	Case2	17	17	17	17	17	17	0	0	0	1.00	1.00	1.00	1.00	1.00	1.00
Pull up Field	Case1	10	10	10	8	9	10	1	9	0	0.88	0.50	**1.00**	0.80	0.90	**1.00**
	Case2	5	5	5	4	4	5	0	2	0	1.00	0.67	1.00	0.80	0.80	1.00
Pull up Method	Case1	10	10	10	10	10	10	1	9	0	0.90	0.52	1.00	1.00	1.00	1.00
	Case2	4	4	4	4	4	4	0	0	0	1.00	1.00	1.00	1.00	1.00	1.00
Push down Field	Case1	10	10	10	10	10	10	10	10	0	0.50	0.50	1.00	1.00	1.00	1.00
	Case2	6	6	6	6	6	5	0	0	0	1.00	1.00	1.00	1.00	1.00	0.83
Push down Method	Case1	10	10	10	10	10	10	1	1	0	0.90	0.90	1.00	1.00	1.00	1.00
	Case2	3	3	3	3	3	2	0	0	0	1.00	1.00	1.00	1.00	1.00	0.67
Inline Superclass	Case1	10	10	10	7	10	10	3	13	2	0.77	0.43	**0.83**	0.70	1.00	1.00
	Case2	7	7	7	5	7	7	1	1	2	0.87	0.87	0.78	0.71	1.00	1.00
Flatten Hierarchy	Case1	10	10	10	0	10	8	0	11	0	0.00	0.48	**1.00**	0.00	1.00	0.80
	Case2	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-
Extract SuperClass Field	Case1	10	10	10	10	8	10	2	0	0	0.83	1.00	1.00	1.00	0.80	1.00
	Case2	2	2	2	2	2	2	1	0	0	1.00	1.00	1.00	1.00	1.00	1.00
Extract SuperClass Method	Case1	10	10	10	10	8	10	1	1	0	0.90	0.90	**1.00**	1.00	0.80	1.00
	Case2	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-
Extract SubClass Field	Case1	10	10	10	10	10	10	2	10	2	0.83	0.50	0.83	1.00	1.00	1.00
	Case2	3	3	3	3	3	2	0	2	0	1.00	0.67	1.00	1.00	1.00	0.67
Extract SubClass Method	Case1	10	10	10	10	10	10	1	10	0	0.90	0.50	**1.00**	1.00	1.00	1.00
	Case2	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-
Replace Class with Field	Case1	10	10	10	10	10	9	0	0	0	1.00	1.00	1.00	1.00	1.00	0.90
	Case2	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-
Overall	Case1	210	210	210	172	205	204	33	221	4	0.83	0.48	**0.98**	0.82	0.98	**0.98**
	Case2	106	106	106	102	102	103	6	50	2	0.94	0.67	**0.98**	0.96	0.96	**0.97**

vant operations, which have a negative impact on the precision value.

V. RELATED WORK

To obtain high-level operations, the most direct way is to track the operations while they are executed, which requires support from the model management tools and the operations be well-defined. Operation recorder implemented in [28] defines operations from low-level to high-level ones, while provides an operation metamodel that all operations conforms to. As a operation-based tool, it can capture more accurate information about operations than state-based tools since the operations are recorded when executed. However, this approach is only meant for EMF models [29] and still under development. The by-example operation recorder [30] provides support for user-specific operations, especially for high-level operations. The precondition of an operation is specified first by designing an example model, after which a list of primitive operations are executed upon this model and the postcondition can be extracted. The characteristics of this high-level operation then, can be calculated from the pre- and postcondition, i.e. the target elements this operation are operated on and how it can change these elements. The primitive operations are not recorded directly, which means the accuracy may suffer a little loss.

There are some approaches that detect high-level operations in a posteriori way, i.e. obtain them according to their executing effects. UMLDiff [31] leverages change pattern queries to detect refactorings. The difference between the initial model and the model after changing is obtained first by model comparison, which is represented by a diff model. Then user can query high-level operations out of this diff model. Compared to our approach, UMLDiff does not support an iterative operation detection, which may leave some high-level operation undetected.

Küster *et al.* [32] and Gerth *et al.* [33] derive changes in a hierarchical way for business process models. The models are separated into fragments with the benefit of the single-entry-single-exit structure, with each fragments composed by several activities. Therefore the operations are classified into primitive ones that only manipulate single activity and composite ones that modify a fragment that contains several activities. Operations can be detected once some model elements change providing that the correspondences between elements of two models are given. However, this approach is only applicable for process models. Langer *et al.* [24] employ a posteriori approach to detect both low- and high-level operations in general software models. A diff model is calculated from the pre- and postcondition of a specified operation. By comparing with diff models, potentially expected operations are derived. The final assurance of the operation still relies on

the comparison with pre- and postconditions of the operations to be matched. Unfortunately, this approach does not consider the *Indefinite length problem* and in some case it cannot detect all the operations of a composite operation.

Khelladi *et al.* [25] propose a detection engine of complex changes which can detects all possible candidates,even in case of overlapping changes. However, as we have seen in Table 3, this algorithm had not performed well in some cases. The search-based detection approach [23] employs a heuristic method to detect refactoring from the possible refactoring sets. This approach does not consider the pre- and postcondition of operations; instead it randomly applies some refactoring sets to the initial model, and find the most similar one to the real result model. The efficiency of this method may be low for the generation of refactoring sets is nontrivial. Unfortunately, all of these approaches ignore conflicts between potential composite operations and some irrelevant operations.

VI. CONCLUSION

As the operation-based modeling tools develop, high-level operations are more comprehensible to users and can facilitate the supported functions such as model transformation. In this paper we introduced an approach to iteratively raise the level of operations. Each composite operation is defined by hierarchical structure and low-level operations which match this structure can be lifted to the corresponding composite operation. To evaluate our approach, several contrast experiments have been conducted, which showed that both precision and recall of composite operation detection are improved compared to the EMO and CCDE algorithms. We have implemented our algorithm based on EMFStore to detect composite operations that are applicable for class diagrams in UML models. In the future we plan to plug this tool into more modeler platforms and to apply to more generic models.

REFERENCES

[1] D. C. Schmidt, "Model-driven engineering," *IEEE Comput. Soc.*, vol. 39, no. 2, pp. 25–31, Feb. 2006.

[2] H. Song, D. B. Rawat, S. Jeschke, and C. Brecher, *Cyber-Physical Systems: Foundations, Principles and Applications*. San Mateo, CA, USA: Morgan Kaufmann, 2016.

[3] S. Jeschke, C. Brecher, H. Song, and D. B. Rawat, *Industrial Internet of Things*. Cham, Switzerland: Springer, 2017.

[4] Y. Jiang *et al.*, "Design and optimization of multiclocked embedded systems using formal techniques," *IEEE Trans. Ind. Electron.*, vol. 62, no. 2, pp. 1270–1278, Feb. 2015.

[5] Y. Jiang *et al.*, "Design of mixed synchronous/asynchronous systems with multiple clocks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2220–2232, Aug. 2015.

[6] Y. Jiang *et al.*, "From stateflow simulation to verified implementation: A verification approach and a real-time train controller design," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2016, pp. 1–11.

[7] Y. Jiang, H. Song, R. Wang, M. Gu, J. Sun, and L. Sha, "Data-centered runtime verification of wireless medical cyber-physical system," *IEEE Trans. Ind. Informat.*, to be published.

[8] Y. Jiang *et al.*, "Use runtime verification to improve the quality of medical care practice," in *Proc. 38th Int. Conf. Softw. Eng. Companion*, May 2016, pp. 112–121.

[9] Y. Jiang *et al.*, "Bayesian-network-based reliability analysis of PLC systems," *IEEE Trans. Ind. Electron.*, vol. 60, no. 11, pp. 5325–5336, Nov. 2013.

[10] H. Zhang, Y. Jiang, W. N. N. Hung, X. Song, M. Gu, and J. Sun, "Symbolic analysis of programmable logic controllers," *IEEE Trans. Comput.*, vol. 63, no. 10, pp. 2563–2575, Oct. 2014.

[11] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer, "An introduction to model versioning," in *Formal Methods for Model-Driven Engineering*. Berlin, Germany: Springer, 2012, pp. 336–398.

[12] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 232–282, Jun. 1998.

[13] P. Langer, T. Mayerhofer, and G. Kappel, "Semantic model differencing utilizing behavioral semantics specifications," in *Proc. Int. Conf. Model Driven Eng. Lang. Syst.*, Sep. 2014, pp. 116–132.

[14] M. Koegel, M. Herrmannsdoerfer, O. von Wesendonk, and J. Helming, "Operation-based conflict detection," in *Proc. 1st Int. Workshop Model Comparison Pract.*, Jul. 2010, pp. 21–30.

[15] C. Schneider and A. Zündorf, and J. Niere, "CoObRA—A small step for development tools to collaborative environments," in *Proc. Workshop Directions Softw. Eng. Environ.*, 2004, pp. 21–28.

[16] B. Bruegge, O. Creighton, J. Helming, and M. Kögel, "Unicase—An ecosystem for unified software engineering research tools," in *Proc. 3rd IEEE Int. Conf. Global Softw. Eng. (ICGSE)*, 2008, pp. 2–17.

[17] M. Köegel and J. Helming, "EMFStore: A model repository for EMF models," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, vol. 2. May 2010, pp. 307–308.

[18] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt, "Adaptability of model comparison tools," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Sep. 2012, pp. 306–309.

[19] P. Brosch *et al.*, "An example is worth a thousand words: Composite operation modeling by-example," in *Proc. Int. Conf. Model Driven Eng. Lang. Syst.*, Oct. 2009, pp. 271–285.

[20] K. Altmanninger and A. Pierantonio, "A categorization for conflicts in model versioning," *e & i Elektrotech. Informationstech.*, vol. 128, no. 11, pp. 421–426, Dec. 2011.

[21] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, Mar. 2006.

[22] S. D. Vermolen, G. Wachsmuth, and E. Visser, "Reconstructing complex metamodel evolution," in *Proc. Int. Conf. Softw. Lang. Eng.*, Jul. 2011, pp. 201–221.

[23] A. ben Fadhel, M. Kessentini, P. Langer, and M. Wimmer, "Search-based detection of high-level model changes," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2012, pp. 212–221.

[24] P. Langer *et al.*, "*A posteriori* operation detection in evolving software models," *J. Syst. Softw.*, vol. 86, no. 2, pp. 551–566, Feb. 2013.

[25] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais, "Detecting complex changes during metamodel evolution," in *Proc. Int. Conf. Adv. Inf. Syst. Eng.*, Jun. 2015, pp. 263–278.

[26] Z. Zhang, R. Zhang, and Z. Qin, "Composite-level conflict detection in UML model versioning," *Math. Problems Eng.*, vol. 2015, 2015, Art. no. 650748.

[27] R. Hebig, D. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *IEEE Trans. Softw. Eng.*, to be published.

[28] M. Herrmannsdoerfer and M. Köegel, "Towards a generic operation recorder for model evolution," in *Proc. 1st Int. Workshop Model Comparison Pract.*, Jul. 2010, pp. 76–81.

[29] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Upper Saddle River, NJ, USA: Pearson Education, 2008.

[30] P. Brosch, M. Seidl, K. Wieland, M. Wimmer, and P. Langer, "The operation recorder: Specifying model refactorings by-example," in *Proc. 24th ACM SIGPLAN Conf. Companion Object Oriented Program. Syst. Lang. Appl.*, Oct. 2009, pp. 791–792.

[31] Z. Xing and E. Stroulia, "Refactoring detection based on UMLDiff change-facts queries," in *Proc. 13th Working Conf. Reverse Eng. (WCRE)*, vol. 6. 2006, pp. 263–274.

[32] J. M. Küster, C. Gerth, A. Förster, and G. Engels, "Detecting and resolving process model differences in the absence of a change log," in *Proc. Int. Conf. Bus. Process Manage.*, Sep. 2008, pp. 244–260.

[33] C. Gerth and J. M. Küster, M. Luckey, and G. Engels, "Detection and resolution of conflicting change operations in version management of process models," *Softw. Syst. Model.*, vol. 12, no. 3, pp. 517–535, Jul. 2013.

RENWEI ZHANG received the M.S. degree in software engineering from Peking University, Beijing, China, in 2012. He is currently pursuing the Ph.D. degree with the School of Software Engineering, Tsinghua University, Beijing. His research interests are software testing, model-driven engineering, and model versioning.

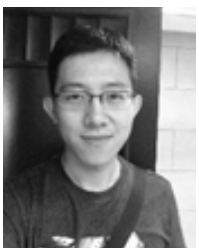
ZHENG QIN is currently a Doctoral Supervisor, a Professor, and the Director of Software Engineering and Management Research Institute and Information Institute, Tsinghua University. He is also a Professor of Tsinghua University and Xian Jiaotong University, and also a part-time Professor of many other high-education organizations, including the Ministry of the National Supervision. He is a member of the Ministry of Education E-Commerce Specialty Teaching Guidance Committee. He is the evaluation expert of the National Science and Technology Award, the Ministry of Education Technology Award, the Major State Basic Research Development Program (973), the National High Technology Research and Development Program of China (863), the National Defense 10th 5-Year Plan, and the Ministry of Education College Undergraduate Teaching. He is a Guest Researcher of the Shanxi Academy and the Henan Academy, and an Editor of the *International Journal of Plant Engineering and Management(E)* and *Journal of E-Business*.

HOUBING SONG (M'12–SM'14) received the Ph.D. degree in electrical engineering from the University of Virginia, Charlottesville, VA, in 2012.

In 2007, he was an Engineering Research Associate with the Texas A&M Transportation Institute, College Station, TX, USA. In 2012, he joined the Department of Electrical and Computer Engineering, West Virginia University, Montgomery, WV, USA, where he is currently an Assistant Professor and the Founding Director of the Security and Optimization for Networked Globe Laboratory and the West Virginia Center of Excellence for Cyber-Physical Systems sponsored by the West Virginia Higher Education Policy Commission. He is an Editor of four books, including *Smart Cities: Foundations, Principles and Applications* (Hoboken, NJ, USA: Wiley, 2017), *Security and Privacy in Cyber-Physical Systems: Foundations, Principles and Applications* (Chichester, U.K.: Wiley, 2017), *Cyber-Physical Systems: Foundations, Principles and Applications* (Waltham, MA, USA: Elsevier, 2016), and *Industrial Internet of Things: Cybermanufacturing Systems* (Cham, Switzerland: Springer, 2016). He has authored more than 100 articles. His research interests include cyber-physical systems, Internet of Things, cloud computing, big data analytics, connected vehicle, wireless communications and networking, and optical communications and networking.

Dr. Song is a member of ACM. He received the Golden Bear Scholar Award, the Highest Faculty Research Award at the West Virginia University Institute of Technology, in 2016.

SHENGNAN LI received the B.S. degree in software engineering from Nankai University, Tianjin, China, in 2011. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. Her research interests are wireless sensor network, indoor localization, and mobile sensing.



XIAO YANG received the M.S. degree from the Department of National University of Defense Technology, China, in 2011. He is currently pursuing the Ph.D. degree in software engineering from Tsinghua University, China. His research interests include intelligent transportation systems, pedestrian simulation, and pattern recognition.