International Journal of Software Engineering and Knowledge Engineering

PRESERVATION OF CLASS INVARIANTS IN REFACTORING UML MODELS

THI-HUONG DAO
VNU, University of Engineering and Technology
144 Xuan Thuy, Cau Giay, Ha Noi, Vietnam
huongdt.di12@vnu.edu.vn

XUAN-TRUONG NGUYEN
Hanoi Pedagogical University No 2
Vinh Phuc, Vietnam
nguyenxuantruong@hpu2.edu.vn

NINH-THUAN TRUONG
VNU, University of Engineering and Technology
144 Xuan Thuy, Cau Giay, Ha Noi, Vietnam
thuantn@vnu.edu.vn

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

In the field of software engineering, *class invariants* is known as a valuable term employed to delineate the semantic of UML class diagram elements (attributes and relationships) and must be held throughout the life-time of instances of the class. *Refactoring*, the activities of re-distributing classes, attributes and methods across the class hierarchy, is a powerful technique that use to improve the quality of software systems. Performing refactoring on UML class diagrams obviously requires a special investigation of invariant-preserving on the refactored models. In this paper, we propose an approach to preserve class invariants in refactoring UML models. In order to achieve this aim, we first formalize the class diagram along with class invariants by mathematical notations. We then constitute the rules for five refactoring operations (deal with class hierarchies) in such a way to guarantee class invariants as well as proving correctness of the refactoring rules. Finally, the paper also makes provision of the proposed approach for practical applications in software re-engineering development process.

Keywords: Class invariants; refactoring UML models; invariant-preserving.

1. Introduction

During the manipulation, the software system has not only to maintain, but also to evolve as its obvious intrinsic properties. This evolution is performed for the sake of improving the quality of software models such as extensibility, modularity,

re-usability, etc. One of the most common techniques which is widely adopted in this process is re-engineering and/or refactoring software codes or models.

Refactoring, first introduced by Opdyke in his thesis, "*is the process of changing a software system in such a way that it does not alter the external behavior, yet improves its internal structure*" [16]. The nature of this process is to re-distribute classes, attributes and methods around the inheritance relationship in order to facilitate future adaptations and extensions.

Class invariants are defined as an assertion that represents the conditions (constraints) of the attributes, which must be true at every stable point in time during the life of an object [17]. The characteristics of class invariants are as follows:

- executing as an additional constraint for pre and post-conditions that appropriate to all public methods for every instance of the class;
- respecting the initial value of a constructor;
- being kept by the public operations of the class.

Consequently, class invariants plays an essential role in describing the precisely semantics of a UML class diagram in a specific context. Therefore, any activity that makes a difference to the software model must take into account the transforming of these invariants.

The process of refactoring on UML models will obviously alter their static structure (e.g., class diagram, object diagram, etc.) as well as dynamic behavior (e.g., state chart diagram, sequence diagram, etc.). In this paper, we specially concentrate on the changing of the static aspect of the UML models, especially on class diagrams and their invariants as well. These transformations may make the model quality become worse, such as class invariants of the initial model is violated. This, for software developers, is one of the big challenges that has to face out.

Basing on classification criteria of complexity, Opdyke [16] showed a list of refactorings that are categorized into two groups, namely *primitive refactorings* and *composite refactorings*. The first kind of refactorings usually refers to quite simple activities such as *MoveAttribute, MoveAssociationEnd, PushDownAttribute, RenameAttribute, ExtractClass, ExtractSuperclass*, *PullUpMethod, etc.*. These activities are considered elementary behavior-preserving transformations in refactoring. The remaining group relates to complex transformations as a composition of primitive refactorings [4], namely *chaining* and *set iteration*. In other words, a *composite refactoring* is usually built from a set of *primitive refactorings* for the sake of exposing more complex behavior-preserving transformations that are more meaningful to the user.

A catalog of refactorings is also represented by in Fowler *et al.* [7] and is organized as follows: *composing methods, moving features between objects, organizing data, simplifying conditional expressions, and dealing with generalizations*. Our aims in this paper are to tackle the problem of how to maintain class invariants in refactoring UML class diagram. Therefore, these activities make some composite refactorings, mostly dealing with *moving methods and attributes around a hierarchy*

of inheritance or *creating new classes* [16]. Concerning this problem, we concentrate on constructing the transformation rules that relate to generalizing refactoring operations, namely *Folding, Abstraction, Composition, Factoring* and *Unfolding*.

Several approaches have been proposed to refactor UML models which using OCL contracts [18], graphical formal modeling notation as named UML-B [14], and model-to-model (M2M) transformations [22]. Their research results, however, are only taken refactoring class diagrams into account in a informal or semi-formal way *without having an extensive view of preservation of class invariants*. Therefore, we propose in this paper an approach to preserving class invariants in refactoring UML models. The main contributions of this paper are (1) *formalizing the UML class diagram along with their class invariants by using mathematical notations*, (2) *defining the template that makes use for describing refactoring operation*, (3) *constructing the rules for refactoring operations in such a way that preserve the class invariants and proving correctness of them as well*, and (4) *presenting the provision of the proposed approach for practical applications in software re-engineering development process*.

The rest of the paper is organized as follows. Section 2 gives an overview of the object oriented models as well as some refactoring operations. We formalize the UML class diagrams along with their invariants by mathematical notations as the foundation theory for transformation rules in Section 3. Section 4 presents some works related to our research. Finally, Section 5 concludes the paper and gives some directions for future works.

2. Refactoring with object-oriented model

In this section, we first present some background knowledge of the object oriented model. Then, some refactoring operations that deal with inheritance in refactoring UML class diagrams are depicted in the overview.

2.1. *The Object-Oriented Model*

An object oriented model is a logic organization of the real world objects (entities) along with constraints on them as well as the relationships among objects [3]. An object-oriented model is casually represented by an UML class diagram (static structure of the object oriented model) and is composed of the following essential concepts:

(1) **Class**: is a means of grouping all the objects which share the same set of *attributes* and *methods*. An object must belong to only one class as an instance of that class. A class is similar to an abstract data type or may also be a primitive type (no attributes), e.g., integer, string, boolean.

- Attributes: represent useful information of a instance of a class (the set of values for the attributes of the object are defined as *state*).
- Methods: define class *behavior* (the set of methods which operate on the

state of the object).

(2) **The relationship among classes**: In the object oriented model, classes are interacted with each other in particular ways that include various types of logical connections. In this paper, we are interested in some followed relation types:

- Association: is a broad term that encompasses just about any logical connection or relationship between classes.
- Aggregation: refers to the formation of a particular class as a result of one class being aggregated or built as a collection.
- Composition: is the composition relationship which is very similar to the aggregation relationship, with the only difference being its key purpose of emphasizing the dependence of the contained class to the life cycle of the container class. It means that the contained class will be obliterated when the container class is destroyed.
- Realization: denotes the implementation of the functionality defined in one class by another class.
- Inheritance: derives a new class (subclass) from an existing class (superclass). The subclass inherits all the attributes and methods of the existing class and may have additional attributes and methods. Inheritance enables you to create new classes that reuse, extend, and modify the behavior that is defined in other classes.
- Class Hierarchy: the class hierarchy defines the inheritance relationship between the classes.

(3) **Constraint of a class**: represents for some conditions, restrictions or assertions which relate to elements of a class (attributes, methods) [9]. Constraint is usually specified by a Boolean expression which must evaluate to a *true* or *false* and must be satisfied (i.e., evaluated to true) by a correct design of the system. Constraints that related to class diagram are classified into three categories as follows:

- Class invariants: is a condition which defines on class attributes and every instance of the class must satisfy;
- Pre-conditions of a method: is a condition that method has to satisfy when it begins to execute;
- Post-conditions of a method: is a condition that method has to satisfy after executing.

Roughly speaking, an object oriented model includes multiple classes along with their relationships as well as constraints on them.

2.2. *Refactoring Class Hierarchies*

Class diagram commonly serves as an effective graphical means to represent an object-oriented model due to the compatibility of characteristics between them. In

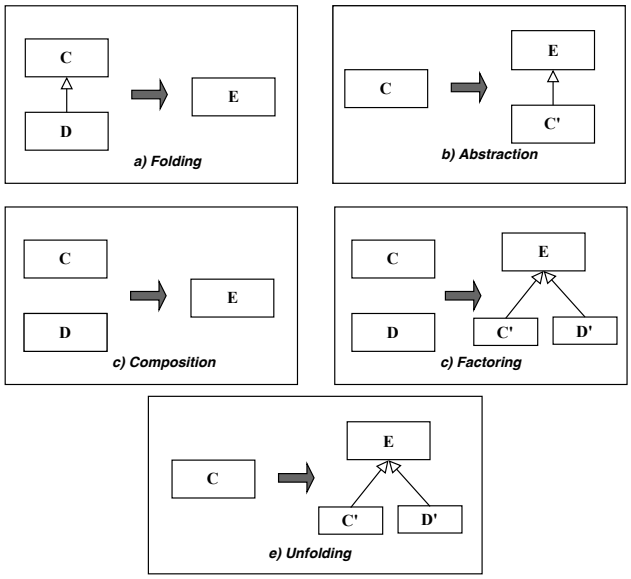

Fig. 1: Refactoring class hierarchies

this subsection, in order to demonstrate the refactoring process related to generalization in object-oriented model, we will describe a set of operations that deal with inheritance in refactoring UML class diagrams, namely Folding, Abstraction, Composition, Factoring and Unfolding [18] as shown in Fig. 1. In general, these operations are described as follows:

(1) *Folding operation*: In the case of two classes which have a direct inheritance relationship, nevertheless, there is no particular interest in the behavior of a base class, either because it is an abstract class or because the amount of operations of the class does not justify having another level in the hierarchy. When that *Folding* operation joins these classes into one for the sake of reducing the level of a class hierarchy.
(2) *Abstraction operation*: If a class with a long list of attributes and methods that make it difficult to reuse as well as maintain the software model. *Abstraction* operation then creates a new base class that can abstract the more general behavior than before one and construct a direct inheritance relationship between them.
(3) *Composition operation*: The multiple inheritances in object oriented model is currently avoided because of the issue with clashes resolving which parent method is being called, etc. *Composition* operation addresses this problem by gathering two classes without inheritance relationship to each other in a new one which groups their behaviors.
(4) *Factoring operation*: One of the purposes of object oriented model is to elim-

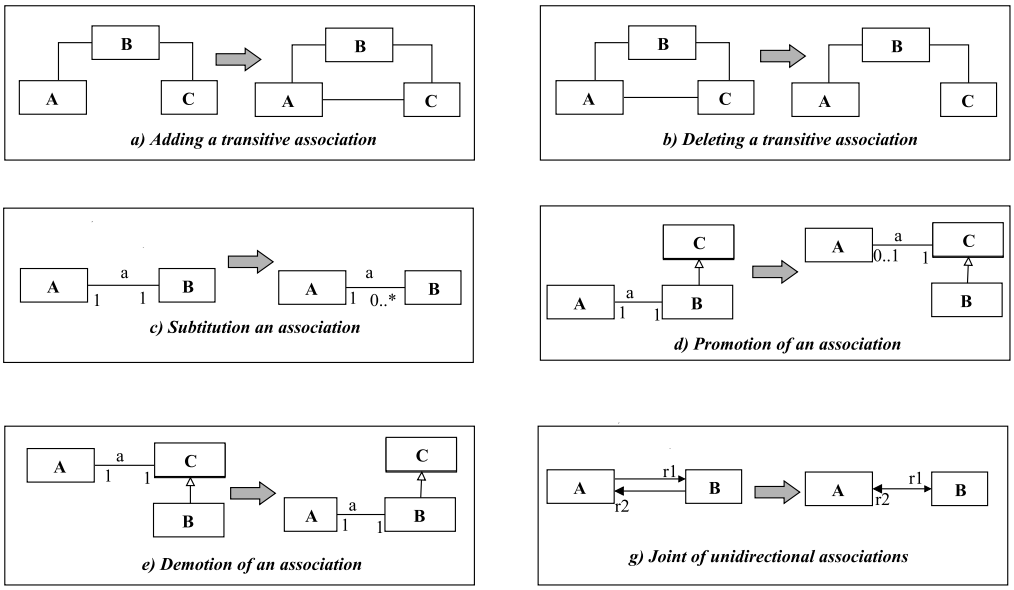

Fig. 2: Refactoring class associations

inate duplicated methods and attributes. Therefore, the *Factoring* operation accomplishes this goal by factoring equivalent methods and attributes in a new base class starting from two classes without inheritance relationship to each other.

(5) *Unfolding operation*: When methods of a class do not refer simultaneously to all the attributes, but only make reference to some of them. The Unfolding operation divides the behavior of this class, generating two classes which maintain a direct inheritance relationship. Such classes arise from carrying out a partition of the attributes in two disjoint subsets

2.3. *Refactoring Class Associations*

Along with the change of attributes and methods of the classes during refactoring is the altering of associations. In this subsection, we describe a summary the results of previous research that relate to the change of associations in refactoring process and are depicted in Fig. 2.

(1) *Adding a transitive association:* Let two given associations, between classes A and B, between classes B and C, respectively. At that, an association may be derived between A and C, determining the appropriate association type, the multiplicities and the navigability of each association end [23].

(2) *Deleting a transitive association:* Given an association between classes A and

B, an association between classes B and C, and an association between A and C, the transitive association between A and C may be deleted [23].

(3) *Substitution of an association:* Given an association R, it may be substituted with a less constrained association of the same name, i.e., in any association R, an association end E with multiplicity $MULT1$ may be substituted with an association end E with multiplicity $MULT2$, where $MULT1 \subseteq MULT2$ [5].

(4) *Promotion of an association:* Given an association R with multiplicity $MULT1$ (connected to a class A) and multiplicity $MULT2$ (connected to a class B), if B is a subclass, then R may be promoted to the superclass of B with the condition that its multiplicity with A after the transformation is optional, i.e. $0 \in MULT1$ [5].

(5) *Demotion of an association:* Given an association R with multiplicity $MULT1$ (connected to a class A) and multiplicity $MULT2$ (connected to a class C), if B is a subclass, then R may be demoted to the subclass of B [5].

(6) *Joint of unidirectional associations:* Two unidirectional associations with navigability in opposite direction may be joined in a plain bidirectional one [8].

3. An approach to invariant-preserving in refactoring UML class diagram

An object-oriented model is usually represented by a class diagram in the UML that contains information about the static structure of a system. From the structural perspective, a class diagram consists of two main components, *classes* and *relationships* among them (also known as *associations*) [2]. From the semantic perspective, a class diagram also includes the constraints that attach to both components. Solving the problem of refactoring class diagram in UML models will naturally affect classes and their associations as well. Therefore, tackling the issue of semantic preserving must comprehensively consider the impact on both these components. Note that, we have only addressed the altering related to structure of classes in this research and suppose that the associations will auto change to fit the evolution model. It means that associations are always validated during the refactoring process.

In this Section, we first introduce the formal representation (mathematically) of model elements along with their semantic constraint definitions. We then describe the structure of a refactoring operation as well as the rules that employing for the refactoring process. We continue to prove the correctness of the refactoring rules. Finally, a small experiment is depicted as an initial guide for developers in performing practical application.

3.1. *Formal representation of a UML class diagram*

Definition 1 (UML class diagram - Model). A UML class diagram, also called a model $\mathcal{M}$, is a 2-tuple $\langle \Sigma_C, \Sigma_A \rangle$, where Σ_C is a set of classes and Σ_A is a set of associations.

Definition 2 (Class). A class $C \in \Sigma_C$ is represented by a 3-tuple $C = \langle N_C, M_C, A_C \rangle$, where N_C is the name of the class, M_C is a set of methods and A_C is a set of attributes.

Definition 3 (Method). A method $m_C^i \in M_C$ is represented by a 3-tuple $m_C^i = \langle N_{M_C}^i, P_{M_C}^i, R_{M_C}^i \rangle$, where $N_{M_C}^i$ is the name of the method, $P_{M_C}^i$ is a set of parameters and $R_{M_C}^i$ is the return type.

Definition 4 (Attribute). An attribute $a_C^i \in A_C$ is represented by a 3-tuple $a_C^i = \langle N_{A_C}^i, T_{A_C}^i, P_{A_C}^i \rangle$, where $N_{A_C}^i$ is the name of the attribute, $T_{A_i}^i$ is the attribute type and $P_{A_C}^i$ is a predicate that represents for attribute constraint.

Definition 5 (Class invariants). An invariant INV_C of the class C is described by the conjunction of the predicates of all the related attributes in the set A_C.

Let $P_{A_C}^i$ is the predicate that represents for the constraint of the attribute $a_C^i \in A_C$ (in the case the attribute a_C^i that has no constraint, the predicate $P_{A_C}^i$ will be assigned by *true* value) and $|A_C| = n$ (the number of elements of the set A_C), the invariants of the class C is depicted by the formula $INV_C = \bigwedge_{i=1}^{n} P_{A_C}^i$.

Definition 6 (Model invariants). An invariant $\mathcal{F}$ of the model $\mathcal{M}$ is defined by the conjunction of all the class invariants that are comprised in this model.

For simplicity, let a model $\mathcal{M}$ which has two classes C and D, the invariants of the model $\mathcal{M}$ is depicted by the formula $\mathcal{F} = INV_C \bigwedge INV_D$, where INV_C and INV_D are the invariants of classes C and D, respectively.

Definition 7 (Association). An association $as \in \Sigma_A$ is represented by a 4-tuple $as = \langle N_{as}, E_{as_1}, E_{as_2}, NAV \rangle$, where $N_{as}, E_{as_1}, E_{as_2}, NAV$ are the names of association, the association end 1, association end 2 and the navigability of this association, respectively.

Definition 8 (Association End). An Association End E_{AS_i} is represented by a 2-tuple $E_{AS_i} = \langle N_{CL}, MULT_i \rangle$, where $i \in [1, 2]$ and $N_{CL}, MULT_i$ are the name of class and the multiplicity of this association end, respectively.

Definition 9 (Semantic Equivalent Methods). Two given methods $m_C^i \in M_C$ of the class C and $m_D^j \in M_D$ of the class D. m_C^i and m_D^j are called semantic equivalence that is denoted by $M_C^i \cong M_D^j$ if only if:

$$\begin{cases} N_{M_C}^i \equiv N_{M_D}^j & \textit{//duplicate names} \\ P_{M_C}^i \equiv P_{M_D}^j & \textit{//duplicate parameters} \\ R_{M_C}^i \equiv R_{M_D}^j & \textit{//duplicate types} \end{cases}$$

Then $M_C \cup M_D$ are divided into equivalence classes denoted by $\mathcal{M}_{C,D} = \{m_{C,D}^{ij}\}$, where:

$$m_C^i, m_D^j \in m_{C,D}^{ij} \iff \begin{cases} m_C^i \in M_C \\ m_D^j \in M_D \\ M_C^i \cong M_D^j \end{cases}$$

We denote a set of m_C^i which is in M_C and not in arbitrary $m_{C,D}^{ij}$ by the notation $M_C \setminus \mathcal{M}_{C,D}$.

Definition 10 (Composable Attributes). Two given attributes $a_C^i \in A_C$ of the class C and $a_D^j \in A_D$ of the class D. a_C^i and a_D^j are called composation that denoted by $A_C^i \cong A_D^j$ if only if:

$$\begin{cases} N_{A_D}^i \equiv N_{A_D}^j & //\textit{duplicate names} \\ T_{A_D}^i \equiv T_{A_D}^j & //\textit{duplicate types} \end{cases}$$

and $P_{A_C}^i$ and $P_{A_D}^j$ cannot be a coincidence.

Then $A_C \cup A_D$ are divided into composation classes denoted by $\mathcal{A}_{C,D} = \{a_{C,D}^{ij}\}$, where:

$$a_C^i, a_D^j \in a_{C,D}^{ij} \iff \begin{cases} a_C^i \in A_C \\ a_D^j \in A_D \\ A_C^i \cong A_D^j \end{cases}$$

We denote a set of a_C^i which is in A_C and not in arbitrary $A_{C,D}^{ij}$ by the notation $A_C \setminus \mathcal{A}_{C,D}$.

As such, the refactoring process is defined through operational refactoring as follows:

Definition 11 (Refactor). A refactor $\mathcal{R}$ is denoted by:

$$\mathcal{R} : \mathcal{M} \xmapsto{OP_{name}} \mathcal{M}'$$

where $\mathcal{M}$ and $\mathcal{M}'$ are the original model and its evolution, respectively, OP_{name} is the applied operational refactoring name.

Definition 12 (Preservation of Class Invariants in Refactoring). A refactor is said to be preservation of class invariants if with any refactoring operation execution, the refactored model invariants are satisfied the original model invariants and the restriction of refactored model invariants to original model is equal to the initial model invariants.

Formally, let $\mathcal{R} : \mathcal{M} \xmapsto{OP_{name}} \mathcal{M}'$ be a refactor, $\mathcal{R}$ is called the preservation of class invariants if:

$$\begin{cases} \mathcal{F} \implies \mathcal{F}' \\ \mathcal{F}'|_{\mathcal{M}} = \mathcal{F} \end{cases}$$

Where $\mathcal{F}$ and $\mathcal{F}'$ are the predicate formulas that describe invariants of the original and refactored models, respectively. In this case, if $\mathcal{F} \implies \mathcal{F}'$, then the values of attribute after refactoring are still in expected range and $\mathcal{F}'|_{\mathcal{M}} = \mathcal{F}$ guarantees that the refactored model preserves the invariants of the initial model.

3.2. *The structure of a refactoring operation*

Before going to clarify the transformation steps in refactoring UML models. We describe the structure of each refactoring operation which is composed of five elements, specially *(1) Operation name*, *(2) Applied situation*, *(3) Original model*, *(4) Evolution model*, and *(5) UML representation*, as shown in Table 1.

Table 1: The structure of a Refactoring operation.

Identifier	**Explanation**
Operation Name	Describing the name of a refactoring operation
Applied situation	Describing the typical situation where applying a refactoring operation
Original model	Describing the initial model
Evolution model	Describing the result model after applying a refactoring operation
UML representation	Giving a UML graphical notations

3.3. *The refactoring rules*

In this paper, we only focus on preserving of class invariants (the constraints on attributes) during the refactoring process. Therefore, we assume that all refactored methods meet the requirements for the behavioral preservation (the pre/post-conditions of the methods are preserved).

As stated above, a composite refactoring is built from some primitive refactorings. Furthermore, these primitive refactorings have been proved the preservation of semantic in [10]. If we make use of these primitive refactorings, we will assume that all of them are validated. In addition, we also make some following assumptions when applying refactoring operations to the original model:

- The names of the methods in the different classes but the same function must be the same (if the methods have the same function but different names, we will proceed to rename these methods in order to make them the same);
- The names of the attributes in the different classes but the same semantics must be the same (if the attributes have the same semantic but different names, we will proceed to rename these attributes in order to make them the same).

Refactoring is referred to as complex work that involves a lot of different activities. The refactorings activities perform on class diagram can be classified into five basic operations that comprising *addition/removal/move* the features (attributes, methods or associations), *generalization* and *specialization* [21]. In this paper, we restrict the process of refactorings on the two last actions use the generalization relationship to transfer elements up and down a class hierarchy, namely *Folding,*

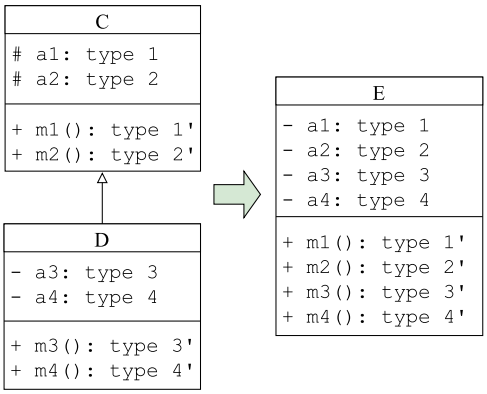

Fig. 3: Folding operation

Abstraction, Composition, Factoring and *Unfolding*. Note that, our purpose is to make *refactoring rules*, the formulas that determine class invariants of the refactored model, for each refactoring operation in order to construct the evolution model in such a way that preserve the invariants of initial model. These refactoring operations are clarified sequentially according to the template described in Subsection 3.2.

(1) **The Folding operation**

- Operation name: *Folding*
- Applied situation:
  - components: let two classes C and D which have a direct inheritance relationship;
  - rational: the behavior of the base class C has no received interest or the amount of methods of the class does not justify having another level in the hierarchy. So the developers want to reduce the level of a class hierarchy in those above cases;
  - performance: joint two classes C and D into new class E that gathering the behavior of both.
- Original model $\mathcal{M}$:
  - base class $C = \langle N_C, M_C, A_C \rangle$;
    - $M_C = \{m_C^1, m_C^2, ..., m_C^n\}$
    - $A_C = \{a_C^1, a_C^2, ..., a_C^m\}$
    - $INV_C = \bigwedge_{i=1}^{m} P_{A_C}^i$, where $P_{A_C}^i$ is the constraint of a_C^i.
  - derived class $D = \langle N_D, M_D, A_D \rangle$;
    - $M_D = \{m_D^1, m_D^2, ..., m_D^p\}$
    - $A_D = \{a_D^1, a_D^2, ..., a_D^q\}$
    - $INV_D = \bigwedge_{i=1}^{q} P_{A_D}^i$, where $P_{A_D}^i$ is the constraint of a_D^i.
  - $\mathcal{M}_{C,D} = \{m_{C,D}^{ij}\}$ and $|\mathcal{M}_{C,D}| = k \quad (0 \leq k \leq \min(n, p))$

- $\mathcal{A}_{C,D} = \{a^{ij}_{C,D}\}$ and $|\mathcal{A}_{C,D}| = h \quad (0 \leq h \leq \min(m, q))$
- $\mathcal{M}_{C,D} = \emptyset$ and $\mathcal{A}_{C,D} = \emptyset$.

- Evolution model $\mathcal{M}'$:
  - class $E = \langle N_E, M_E, A_E \rangle$, is defined as follows:
    * $M_E = M_C \cup M_D$
    * $A_E = A_C \cup A_D$
    * $INV_E = \bigwedge_{i=1}^{m+q} P^i_{A_E}$, where $P^i_{A_E}$ is the constraint of a^i_E and *determined by the rule*:

$$P^i_{A_E} = \begin{cases} P^i_{A_C} & \text{if } a^i_{A_E} \in A_C \\ P^i_{A_D} & \text{if } a^i_{A_E} \in A_D \end{cases}$$

- UML representation: the original/evolution models for the *Folding* operation as illustrated in Fig. 3. Suppose that class C that contains two attributes a_1, a_2 and two methods m_1, m_2; class D that contains two attributes a_3, a_4 and two methods m_3, m_4 and class D is the descendant of class C. However, class C does not show its role in the inheritance relationship (i.e., C is an abstract class but has only one subclass D). The *Folding* operation unites these classes into new class E that gathers the elements of both.

(2) **The Abstraction operation**

- Operation name: *Abstraction*
- Applied situation:
  - components: let class C which has a long methods and attributes;
  - rational: difficult to reuse and maintain the software models as well;
  - performance: create a new base class E that abstract the more general behavior identified inside another class and create the class C' is the remain part of the class C after extracting the class E, constructed a inheritance relationship between the class E and the class C'.
- Original model $\mathcal{M}$:
  - class $C = \langle N_C, M_C, A_C \rangle$;
    * $M_C = \{m^1_C, m^2_C, ..., m^p_C, ..., m^n_C\} \quad (1 \leq p \leq n)$. Assume that $M^1_C = \{m^1_C, m^2_C, ..., m^p_C\}$ and $M^2_C = \{m^{p+1}_C, m^{p+2}_C, ..., m^n_C\}$
    * $A_C = \{a^1_C, a^2_C, ..., a^q_C, ..., a^m_C\} \quad (1 \leq q \leq m)$.
      Assume that $A^1_C = \{a^1_C, a^2_C, ..., a^q_C\}$ and $A^2_C = \{a^{q+1}_C, a^{q+2}_C, ..., a^m_C\}$.
    * $INV_C = \bigwedge_{i=1}^{m} P^i_{A_C}$, where $P^i_{A_C}$ is the constraint of a^i_C.
- Evolution model $\mathcal{M}'$: we assume without loss of generality that all methods and attributes are pulled up to the new base class E, which are ordered from 1 to p, and 1 to k, respectively.
  A new base class E is the superclass of class C' and is defined as follows:

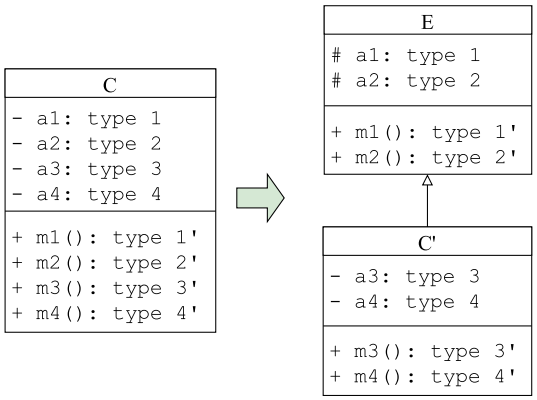

Fig. 4: Abstraction operation

- class $E = \langle N_E, M_E, A_E \rangle$, where:
  - $M_E = M_C^1 = \{m_C^1, m_C^2, ..., m_C^p\}$
  - $A_E = A_C^1 = \{a_C^1, a_C^2, ..., a_C^q\}$
  - $INV_E = \bigwedge_{i=1}^{q} P_{A_E}^i$, where $P_{A_E}^i$ is the constraint of a_E^i and *determined by the rule*: $P_{A_E}^i \equiv P_{A_C^1}^i$.
- Class $C' = \langle N_{C'}, M_{C'}, A_{C'} \rangle$, where:
  - $M_{C'} = M_C^2 = \{m_C^{p+1}, ..., m_C^n\}$
  - $A_{C'} = A_C^2 = \{a_C^{q+1}, ..., a_C^m\}$
  - $INV_{C'} = \bigwedge_{i=1}^{m-q} P_{A_{C'}}^i$, where $P_{A_{C'}}^i$ is the constraint of $a_{C'}^i$ and *determined by the rule*: $P_{A_{C'}}^i = P_{A_C^2}^{i+q}$.

- UML representation: the original/evolution models for the Abstraction operation as illustrated in Fig. 4. Suppose that class C contains four attributes a_1, a_2, a_3, a_4 and four methods m_1, m_2, m_3, m_4, the Abstraction operation extracts attributes a_1, a_2 and methods m_1, m_2 from class C into a new base class E. Class C' is the remain part of class C which has two attributes a_3, a_4 and two methods m_3, m_4. Finally, the inheritance relationship between class E and class C' is created.

(3) **The Composition operation**

- Operation name: *Composition*
- Applied situation:
  - components: let two classes C and D have no inheritance relationship with each other;
  - rational: avoiding multiple inheritances;
  - performance: create the class E that is composed by group as well as gather behavior of both classes C and D.

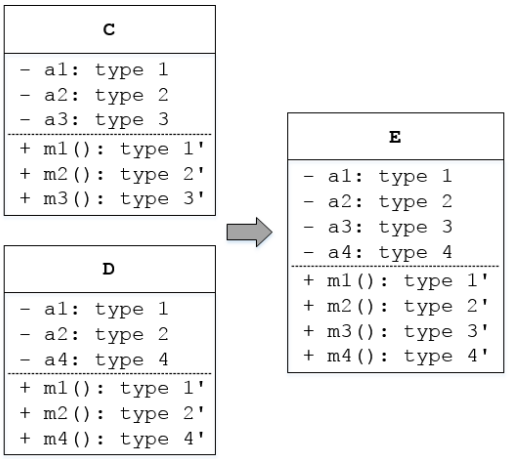

Fig. 5: Composition operation

- Original model $\mathcal{M}$:
  - class $C = \langle N_C, M_C, A_C \rangle$;
    - $M_C = \{m_C^1, m_C^2, ..., m_C^n\}$
    - $A_C = \{a_C^1, a_C^2, ..., a_C^m\}$
    - $INV_C = \bigwedge_{i=1}^{m} P_{A_C}^i$, where $P_{A_C}^i$ is the constraint of a_C^i.
  - class $D = \langle N_D, M_D, A_D \rangle$;
    - $M_D = \{m_D^1, m_D^2, ..., m_D^p\}$
    - $A_D = \{a_D^1, a_D^2, ..., a_D^q\}$
    - $INV_D = \bigwedge_{i=1}^{q} P_{A_D}^i$, where $P_{A_D}^i$ is the constraint of a_D^i.
  - $\mathcal{M}_{C,D} = \{m_{C,D}^{ij}\}$ and $|\mathcal{M}_{C,D}| = k (0 \leq k \leq \min(n, p))$
  - $\mathcal{A}_{C,D} = \{a_{C,D}^{ij}\}$ and $|\mathcal{A}_{C,D}| = h \quad (0 \leq h \leq \min(m, q))$
  - $\mathcal{M}_{C,D} \neq \emptyset$
  - $\mathcal{A}_{C,D} \neq \emptyset$
- Evolution model $\mathcal{M}'$:
  - class $E = \langle N_E, M_E, A_E \rangle$ is defined as follows:
    - $M_E = \{m_E^i : m_E^i \in \mathcal{M}_{C,D}\} \cup (M_C \setminus \mathcal{M}_{C,D}) \cup (M_D \setminus \mathcal{M}_{C,D})$
    - $A_E = \{a_E^i : a_E^i \in \mathcal{A}_{C,D}\} \cup (A_C \setminus \mathcal{A}_{C,D}) \cup (A_D \setminus \mathcal{A}_{C,D})$
    - $INV_E = \bigwedge_{i=1}^{m+q-h} P_{A_E}^i$, where $P_{A_E}^i$ is the constraint of a_E^i and *determined by the rule*:

$$P_{A_E}^i = \begin{cases} P_{A_C}^i \vee P_{A_D}^i & \text{if } a_{A_E}^i \in \mathcal{A}_{C,D} \\ P_{A_C}^i & \text{if } a_{A_E}^i \in (A_C \setminus \mathcal{A}_{C,D}) \\ P_{A_D}^i & \text{if } a_{A_E}^i \in (A_D \setminus \mathcal{A}_{C,D}) \end{cases}$$

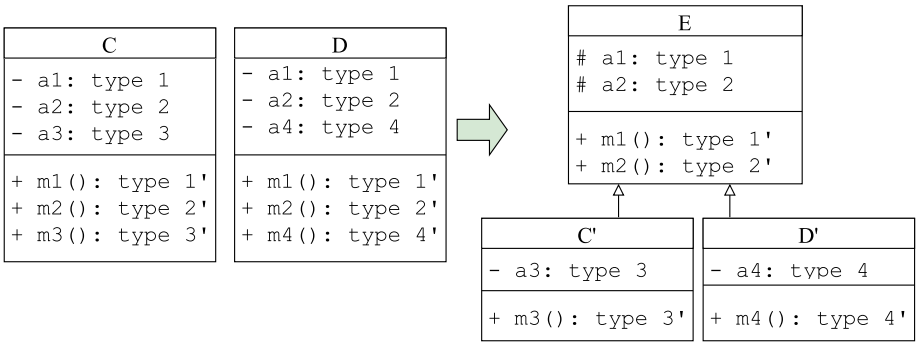

Fig. 6: Factoring operation

- UML representation: the original/evolution models for the Composition operation as illustrated in Fig. 5. Suppose that class C that contains three attributes a_1, a_2, a_3 and three methods m_1, m_2, m_3; class D that contains three attributes a_1, a_2, a_4 and three methods m_1, m_2, m_4; note that, a_1, a_2 are Composable Attributes (Def. 10) and m_1, m_2 are Semantic Equivalent Methods (Def. 9); C and D have without inheritance relationship. The *Composition* operation creates class E that is composed by grouping as well as gathering elements of both classes C and D without duplication of Semantic Equivalent Methods and Composable Attributes.

(4) **The Factoring operation**

- Operation name: *Factoring*
- Applied situation:
  - components: let two classes C and D which have without inheritance relationship to each other but have some semantic equivalent methods and composable attributes as well;
  - rational: eliminate duplicated methods and attributes;
  - performance: create a new base class E that is composed by semantic equivalent methods and composable attributes from the class C and the class D, the classes C' and D' are the remain part of the class C and the class D, respectively. Classes C' and D' have a direct inheritance relationship from class E.
- Original model $\mathcal{M}$:
  - class $C = \langle N_C, M_C, A_C \rangle$;
    - $M_C = \{m_C^1, m_C^2, ..., m_C^n\}$.
    - $A_C = \{a_C^1, a_C^2, ..., a_C^m\}$.
    - $INV_C = \bigwedge_{i=1}^{m} P_{A_C}^i$, where $P_{A_C}^i$ is the constraint of a_C^i.
  - class $D = \langle N_D, M_D, A_D \rangle$;
    - $M_D = \{m_D^1, m_D^2, ..., m_D^p\}$

* $A_D = \{a_D^1, a_D^2, ..., a_D^q\}$
* $INV_D = \bigwedge_{i=1}^{q} P_{A_D}^i$, where $P_{A_D}^i$ is the constraint of a_D^i.

– $\mathcal{M}_{C,D} = \{m_{C,D}^{ij}\}$ and $|\mathcal{M}_{C,D}| = k (0 \leq k \leq \min(n, p))$
– $\mathcal{A}_{C,D} = \{a_{C,D}^{ij}\}$ and $|\mathcal{A}_{C,D}| = h \quad (0 \leq h \leq \min(m, q))$
– $\mathcal{M}_{C,D} \neq \emptyset$ and $\mathcal{A}_{C,D} \neq \emptyset$.

- Evolution model $\mathcal{M}'$:
  – class $C' = \langle N'_C, M_{C'}, A_{C'} \rangle$ is defined as follows:
    * $M_{C'} = \{m_{C'}^i : m_{C'}^i \in (M_C \setminus \mathcal{M}_{C,D})\}$.
    * $A_{C'} = \{a_{C'}^i : a_{C'}^i \in (A_C \setminus \mathcal{A}_{C,D})\} = \{a_C^{h+1}, a_C^{h+2}, ..., a_C^m\}$.
    * $INV_{C'} = \bigwedge_{i=1}^{(m-h)} P_{A_{C'}}^i = \bigwedge_{i=h+1}^{m} P_{A_C}^i$, where $P_{A_{C'}}^i$ is the constraint of $a_{C'}^i$ and *determined by the rule*: $P_{A_{C'}}^i = P_{A_C}^{(i+h)}$, $i \in \{1, 2, ..., (m-h)\}$.
  – Class $D' = \langle N_{D'}, M_{D'}, A_{D'} \rangle$ is defined as follows:
    * $M_{D'} = \{m_{D'}^i : m_{D'}^i \in (M_D \setminus \mathcal{M}_{C,D})\}$
    * $A_{D'} = \{a_{D'}^i : a_{D'}^i \in (A_D \setminus \mathcal{A}_{C,D})\} = \{a_D^{h+1}, a_D^{h+2}, ..., a_D^q\}$
    * $INV_{D'} = \bigwedge_{i=1}^{(q-h)} P_{A_{D'}}^i = \bigwedge_{i=h+1}^{q} P_{A_D}^i$, where $P_{A_{D'}}^i$ is the constraint of $a_{D'}^i$ and *determined by the rule*: $P_{A_{D'}}^i = P_{A_D}^{(i+h)}$, $i \in \{1, 2, ..., (q-h)\}$.
  – class $E = \langle N_E, M_E, A_E \rangle$ is defined as follows:
    * $M_E = \{m_E^i : m_E^i \in \mathcal{M}_{C,D}\}$
    * $A_E = \{a_E^i : a_E^i \in \mathcal{A}_{C,D}\}$
    * $INV_E = \bigwedge_{i=1}^{h} P_{A_E}^i$, where $P_{A_E}^i$ is the constraint of a_E^i and *determined by the rule*: $P_{A_E}^i = P_{A_C}^i \vee P_{A_D}^i$ if $a_{A_E}^i \in \mathcal{A}_{C,D}$
- UML representation: the original/evolution models for the Factoring operation as illustrated in Fig. 6. Suppose that class C that contains three attributes a_1, a_2, a_3 and three methods m_1, m_2, m_3; class D that contains three attributes a_1, a_2, a_4 and three methods m_1, m_2, m_4; note that, a_1, a_2 are Composable Attributes and m_1, m_2 are Semantic Equivalent Methods; C and D have without inheritance relationship. The *Factoring* operation creates class E that is composed by gather Semantic Equivalent Methods (m_1, m_2) and Composable Attributes (a_1, a_2). Classes C' and D' are the remain parts of class C and class D, respectively and both of them are descendants of class E.

(5) **The Unfolding operation**

- Operation name: *Unfolding*
- Applied situation:
  – components: let class C with a long methods which does not reference simultaneously to all attributes, but only make reference to some of them;

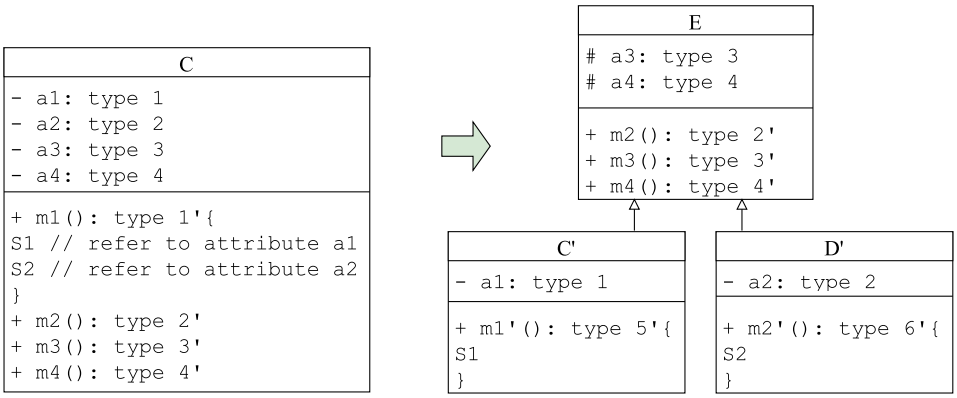

Fig. 7: Unfolding operation

  - rational: optimizing performance of source code;
  - performance: divide the method of a class C, generating multi classes which maintain a direct relationship. Such the classes arise from carrying out a partition of the attributes in many disjoint subsets.

- Original model $\mathcal{M}$:
  - class $C = \langle N_C, M_C, A_C \rangle$;
    - $M_C = \{m_C^1, m_C^2, ..., m_C^n\}$
    - $A_C = \{a_C^1, a_C^2, ..., a_C^p, a_C^{p+1}, ..., a_C^{p+q}, a_C^{p+q+1}, ..., a^(p+q+m)_C\}$. Assume that $A_C^1 = \{a_C^1, a_C^2, ..., a_C^p\}$, $A_C^2 = \{a_C^{p+1}, a_C^{p+2}, ..., a_C^{p+q}\}$ and $A_C^3 = \{a_C^{p+q+1}, a_C^{p+q+2}, ..., a_C^{p+q+m}\}$.
    - $INV_C = \bigwedge_{i=1}^{p+q+m} P_{A_C}^i$, where $P_{A_C}^i$ is the constraint of a_C^i.
  - Without loss of generality we may assume that:
    - the code segments S_1, S_2 belonging to the method m_C^1 refer to the variables $a_C^1, ..., a_C^p$ and $a_C^{p+1}, ..., a_C^{p+q}$, respectively;
    - the method m_C^1 is composed of by just two code segments S_1 and S_2;
    - we take into account only method m_C^1, other methods are done in the similar manner.
- Evolution model $\mathcal{M}'$:
  - class $C' = \langle N_{C'}, M_{C'}, A_{C'} \rangle$ is defined as follows:
    - $M_{C'} = \{m_{C'}^i : m_{C'}^i \quad \text{that is composed by code segment} \quad S_1\}$
    - $A_{C'} = A_C^1 = \{a_C^i : 0 \leq i \leq p\}$
    - $INV_{C'} = \bigwedge_{i=1}^{p} P_{A_{C'}}^i$, where $P_{A_{C'}}^i = P_{A_C}^i$, is the constraint of $a_{C'}^i$.
  - class $D' = \langle N_{D'}, M_{D'}, A_{D'} \rangle$ is defined as follows:
    - $M_{D'} = \{m_{D'}^i : m_{D'}^i \quad \text{that is composed by code segment} \quad S_2\}$
    - $A_{D'} = A_C^2 = \{a_C^i : (p+1) \leq i \leq q\}$

* $INV_{D'} = \bigwedge_{i=p+1}^{p+q} P^i_{A_{D'}}$, where $P^i_{A_{D'}} = P^i_{A_C}$, is the constraint of $a^i_{D'}$.

– class $E = \langle N_E, M_E, A_E \rangle$ is defined as follows:

* $M_E = \{m^i_C : m^i_C \in (M_C \setminus m^1_C)\}$
* $A_E = A^3_C = \{a^i_C : a^i_C \in (A_C \setminus (A_{C'} \cup A_{D'}))\}$
* $INV_E = \bigwedge_{i=p+q+1}^{p+q+m} P^i_{A_E}$, where $P^i_{A_E} = P^i_{A_C}$, is the constraint of a^i_E, and:

– classes C' and D' have a direct inheritance relationship from class E and $A_{C'} \cap A_{D'} = \emptyset$.

- UML representation: the original/evolution models for the Unfolding operation as illustrated in Fig. 7. Suppose that class C with a long method m_1 which includes code segments S_1 and S_2; S_1 refers to a_1 and S_2 refers to a_2. *Unfolding* operation divides m_1 into $m_{1'}$ and $m_{2'}$ methods; creating class C' that contains $m_{1'}$ and a_1, class D' that contains $m_{2'}$ and a_2 and class E is the remain part of class C after extracting method m_1 and attributes a_1, a_2. Classes C' and D' are subclasses of class E.

3.4. *Validation of the proposed refactoring rules*

Subsection 3.1 has sequentially presented the formalized concepts of UML class diagram elements in which we are particularly interested in some key concepts, namely *Class invariants* and *Model invariants*. Furthermore, Def. 12 also defines the concept of *preservation of class invariants in refactoring process*. For the sake of correctness of the proposed approach, we are going to introduce a proposition concerning the preservation of class invariants in refactoring process.

In order to facilitate the next presentation, we denote the set of refactoring operations that are introduced in Subsection 3.3 by the set $OPERATIONS$:

$$OPERATIONS = \{Folding, Abstraction, Composition, Factoring, Unfolding\}.$$

Proposition 1. For all refactors $\mathcal{R} : \mathcal{M} \xmapsto{OP_{name}} \mathcal{M}'$, $OP_{name} \in OPERATIONS$, $\mathcal{R}$ is satisfied all conditions of *preservation of class invariants*.

Proof.

Let $\mathcal{R}$ be a refactor: $\mathcal{M} \xmapsto{OP_{name}} \mathcal{M}'$ and OP_{name} be an arbitrarily chosen element of $OPERATIONS$. We must prove that $\mathcal{R}$ is satisfied all conditions of preservation of class invariants. According to Def. 12, we must prove that:

$$\begin{cases} \mathcal{F} \implies \mathcal{F}' \\ \mathcal{F}'|_{\mathcal{M}} = \mathcal{F} \end{cases}$$

Where $\mathcal{F}$ and $\mathcal{F}'$ are the predicate formulas that describe invariants of $\mathcal{M}$ and $\mathcal{M}'$, respectively.

As previously explained, the key idea of refactoring process is to re-distribute model elements (classes, attributes and methods) around the class hierarchy relationship. Especially, the operations that involve attributes are fallen into two cases as follows:

(1) The first case comprises activities that *just simple perform a re-distribution of discrete attributes* (non-existing of composable attributes in the initial model), such as *Folding*, *Abstraction*, *Unfolding* operations.
(2) The other case is composed of activities that not only *re-distribute discrete attributes* but also *consider uniting composable attributes* in the initial model, such as *Composition*, *Factoring* operations.

For the first case, it is clear that the two conditions of the Def. 12 are easily satisfied because a re-distribution of discrete attributes does not lead to a change their constraints. In other words, the invariants of the refactoring model always preserves the invariants of the original one.

In the second case that involves both a re-distribution of discrete attributes and uniting composable attributes. From the first case, the preservation of class invariants is now tackled only for attributes that satisfy composable conditions.

Refactoring rule of uniting the constraints of composable attributes are shown in Subsection 3.3 as follows:

$$P^i_{A_E} = P^i_{A_C} \vee P^i_{A_D} \text{ if } a^i_{A_E} \in \mathcal{A}_{C,D}. \tag{1}$$

Now, we will prove that the two conditions of Def. 12 will be satisfied by applying the formula (1) in the refactoring process[1].

Let A_C, A_D, INV_C, INV_D be the set of attributes, invariants of classes C and D, respectively and denote by:

$$A_C = \left\{a^1_C, a^2_C, ..., a^m_C\right\} \text{ and } INV_C = \bigwedge_{i=1}^{m} P^i_{A_C}$$
$$A_D = \left\{a^1_D, a^2_D, ..., a^q_D\right\} \text{ and } INV_D = \bigwedge_{i=1}^{q} P^i_{A_D}$$

where $P^i_{A_C}, P^i_{A_D}$ are the predicates that represent for the constraint of the attributes a^i_C, a^i_D, respectively.

Assume that $\mathcal{A}_{C,D} = \{a^{ij}_{C,D}\}$ be the set of Composable Attributes (Def. 10) of the classes C, D; $|\mathcal{A}_{C,D}| = h$ (h is the cardinality of $\mathcal{A}_{C,D}$), where $h \leq \min(m, q)$ and $\mathcal{A}_{C,D} \neq \emptyset$. We call E is the class that contains all of Composable Attributes of both C and D, then $A_E = \left\{a^1_E, a^2_E, ..., a^h_E\right\} = \mathcal{A}_{C,D}$.

Without loss of generality, we reorder the elements of the sets A_C, A_D and A_E

[1] *Due to the combination of conjunction and disjunction operations, we just prove for the case of the initial model that contains two classes, the other cases that initial model has multiple classes will turn into two classes by combination of these classes.*

as follows:

$$A_C = \{a_C^1, a_C^2, ..., a_C^h, a_C^{h+1}, ..., a_C^m\}; INV_C = (\wedge_{i=1}^{h} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i)$$
$$A_D = \{a_D^1, a_D^2, ..., a_D^h, a_D^{h+1}, ..., a_D^q\}; INV_D = (\wedge_{i=1}^{h} P_{A_D}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i)$$
$$A_E = \{a_E^1, a_E^2, ..., a_E^h\}; INV_E = \wedge_{i=1}^{h} P_{A_E}^i,$$

where $P_{A_E}^i$ is defined according to the formula as depicted by (1).

Suppose that the remaining attributes of classes C and D after extracting the composable attributes are held by classes C' and D', respectively. The set of attributes and invariants of classes C', D' are denoted by:

$$A_{C'} = \{a_C^{h+1}, a_C^{h+2}, ..., a_C^m\} \text{ and } INV_{C'} = \bigwedge_{i=h+1}^{m} P_{A_C}^i$$
$$A_{D'} = \{a_D^{h+1}, a_D^{h+2}, ..., a_D^q\} \text{ and } INV'_D = \bigwedge_{i=h+1}^{q} P_{A_D}^i$$

According to Def. 6, we have the invariants of the original model $\mathcal{M}$ as:

$$\begin{aligned} \mathcal{F} &= INV_C \wedge INV_D \\ &= (\wedge_{i=1}^{h} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i) \wedge (\wedge_{i=1}^{h} P_{A_D}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i) \\ &= \wedge_{i=1}^{h} (P_{A_C}^i \wedge P_{A_D}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i) \end{aligned} \tag{2}$$

After refactoring process, we get the invariants of the refactored model $\mathcal{M}'$ as:

$$\begin{aligned} \mathcal{F}' &= INV_E \wedge INV'_C \wedge INV'_D \\ &= (\wedge_{i=1}^{h} P_{A_E}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i) \\ &= \wedge_{i=1}^{h} (P_{A_C}^i \vee P_{A_D}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i) \end{aligned} \tag{3}$$

From the representation of $\mathcal{F}$ by (2) and $\mathcal{F}'$ by (3), the first condition of Def. 12 $\mathcal{F} \implies \mathcal{F}'$ is easily satisfied. Now, we proceed to show that the second condition is also satisfied (notice that, C and D that has no relationship). Indeed, from the Def. 6 we get:

$$\begin{aligned} \mathcal{F}'|_{\mathcal{M}} &= (INV_E \wedge INV'_C \wedge INV'_D)|_{\mathcal{M}} \\ &= ((\wedge_{i=1}^{h} P_{A_E}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i))|_{\mathcal{M}} \\ &= (\wedge_{i=1}^{h} (P_{A_C}^i \vee P_{A_D}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i))|_{\mathcal{M}} \\ &= (\wedge_{i=1}^{h} (P_{A_C}^i \vee P_{A_D}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i))|_C \wedge \\ &\quad \wedge (\wedge_{i=1}^{h} (P_{A_C}^i \vee P_{A_D}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i))|_D \\ &= (\wedge_{i=1}^{h} P_{A_C}^i) \wedge (\wedge_{i=h+1}^{m} P_{A_C}^i) \wedge (\wedge_{i=1}^{h} P_{A_D}^i) \wedge (\wedge_{i=h+1}^{q} P_{A_D}^i) \\ &= INV_C \wedge INV_D \\ &= \mathcal{F} \end{aligned} \tag{4}$$

□

As such, the invariant constraints of the refactored model have been fulfilled. In other words, the refactored model preserves the class invariants of the initial model.

3.5. *Checking Invariant-Preserving in Refactoring UML Class Diagram*

As mentioned before, our objective in this paper is to preserve the class invariants in refactoring process. We have also proved the validation of the refactoring rules by the mathematical method in Subsection 3.4. In practice, the developers may *carry out some other activities on the refactored model* along with the refactoring process (e.g., adding new attributes, modifying existing attributes, etc.). Therefore, we also provide an algorithm that uses to check the invariant-preserving of the refactored model in this case, as shown in Algorithm 1. As such, the refactoring process can be put together with other maintenance activities in a flexible manner for sake of improving model quality in the best way possible.

The checking algorithm takes both models (initial and refactored model) and both formulas $\mathcal{F}$ (initial model invariants) and $\mathcal{F}'$ (refactored model invariants) as inputs and then returns the result of checking invariant-preserving process.

The result of checking algorithm will return *true* or *false*, in which case the two conditions of Def. 12 are fulfilled (it means that, both formulas $\mathcal{F} \implies \mathcal{F}'$ and $\mathcal{F}'|_{\mathcal{M}} = \mathcal{F}$ are satisfied), the checking result will return *true*. In the other cases, the checking result will returns *false*.

Algorithm 1 : Checking Invariant-Preserving in Refactoring UML Class Diagram

Input : $\mathcal{M}(\mathcal{F} : Predicate)$: Model
Input : $\mathcal{M}'(\mathcal{F}' : Predicate)$: Model
Output: invPreserve: Boolean

```
Function invPreserve ← Checking (F, F')
 begin
    if (F ⟹ F') and (F'|_M = F) then
       return true
    end
    return false
end
```

From the experiment perspective, we also make provision of the proposed approach for practical applications as follows.

(1) *Building initial model*: Building an initial model software using UML class diagram as well as considering the semantic constraints of model elements; then establishing the formulas of class invariants (Def. 5) and model invariants (Def. 6).
(2) *Refactoring model*: Using refactoring rules to restructure the model (Subsection 3.3) and re-establishing class and model invariants of refactored model.
(3) *Checking invariant preservation*: Checking the preservation of invariants between the original and the refactored model in order to ensure that the refactoring process have been correctly performed (Algorithm 1).

4. Related Work

We present in this section the state-of-the-art of refactoring UML diagrams, especially in refactoring UML class diagrams.

Wimmer *et al.* [22] provided a catalogue of refactoring for the model to model transformation that is based on a set of rules. Their work is meaningful not only quality attributes related to maintainability such as readability, re-usability, and extensibility of the transformations, but also the performance of transformations and illustrates throughout metamodel annotated with OCL constraints. However, the refactoring catalogue was depicted by the natural language which may be lead to difficulty in automating the refactoring process.

Sunyé *et al.* [21] performed refactoring in UML models, specially class diagrams and state machines. Concerning the class diagram, they just introduced refactoring operations, namely (1) Add feature/association, (2) Remove feature/association and (3) Move element. Concerning the state machines, for each state machine, they are interested in both describing the operations as well as defining OCL constraints that includes pre and post-condions. These constraints were used in refactoring in order to guarantee the preservation of the machine's behavior. However, their work did not mention invariant-preserving in refactoring UML class diagrams.

Other researcher in the trend of model transformation, Ivan Porres [20] presented an action language which resembles OCL that is capable of model transformation. He focused on how to implement refactoring as a collection of transformation rules. He also provided a few refactorings for class models and state machine models and implements them in an experimental tool. How ever, he did not discuss if a refactoring is an actual improvement of a design or if it preserves a given behavioral property of the model.

Thomas and Marković *et al.* [1] developed a few refactorings for class models using QVT. They focused on OCL annotated models so that any changes made by refactoring a model are automatically reflected in OCL constraints. However, they just illustrated the proposed approach along with a simple operation in refactoring (`MoveAttribute`) and did not generalize the semantic preservation in refactoring software models.

Alessandro Folli and Tom Mens [6] developed the Attributed Graph Grammar System (AGG) using graph transformations to execute refactorings in UML models. They paid attention to class models and state machine models and defined a metamodel which is similar to the UML metamodel as a type graph. Their proposed approach has no novelty because of the same way to represent class diagram by the type graph and the UML metamodel.

In [10], Markovic *et al.* formalized and proved the preservation of semantic of some primitive refactorings, namely *MoveAttribute, MoveAssociationEnd, PushDownAttibute, RenameAttribute, ExtractClass, ExtractSuperclass* and *PullUpMethod*. They implemented on class diagram annotated with OCL constraints. The

refactoring process might naturally affect on the syntax of OCL expressions. However, they have shown that even thought the syntactic of the OCL expressions was change, its semantic were preserved in the new refactored models.

Tiago Massoni *et al.* proposed an approach that employ the invariants to make the program syntactically amenable to the desired refactoring, before applying the refactoring itself by using somes primitive program transformations. However, their research was performed at the implementation stage and was considered the basis for the future refactoring process [11].

One of the works that is closest to our research is presented by Claudia Pereira *et al.* [18]. Their work based on the model to model transformations that pay attention at behavior-preserving. Class diagram can be modeled by set theory and refactoring is referred to transforming rules. The advantage of this research's result displays in the initial promotion in modeling a class diagram along with its constraints. However, their research has not been conducted in a thorough way (still using natural language to represent the refactoring operations).

In comparison to prior works, our approach focuses on preservation of class invariants in refactoring UML models. This work different from other in the purpose as well as the way to represent class diagrams (mathematical approach) along with their invariants. Meanwhile other previous works usually based on natural languages and have no an extensive view of preserving class invariants.

5. Conclusion and Future Work

It has been many works to refactor UML model, however, they usually represented class diagram in a semi-formal or an informal ways as well as depicted the transformation rules by the natural language. They haven't yet considered the problem of preserving class invariants during refactoring process. Therefore, our research focuses on finding new approach to face out this problem.

We have proposed in this paper an approach to take into account of class invariants in refactoring UML models. We first formalize the elements of UML class diagram, together with their invariants by making use of mathematical notations. We then introduce five operations that involve the hierarchy relationship. Furthermore, we also propose the refactoring rules in refactoring process and proving that these rules satisfy the conditions of preservation of class invariants.

In addition, we also present a checking invariant-preserving algorithm in case of the developers desire to make features update on the evolution model. This demonstrates the flexibility of the proposed approach when combining with other maintain activities to improve the quality of the software model. This algorithm is the fundamental principle to build a tool that supports for automated verification of class invariant-preserving in refactoring UML model process.

In the future, we will study to utilize the proposed approach for other UML models, e.g., use case, state diagrams, etc. with other constraints of the model, e.g., pre/post-conditions, etc. and illustrate this approach through a practical case study.

We will also consider the combination of these refactoring operations for the sake of making more meaningful jobs for the users.

Acknowledgments

This work is partly supported by the project no. 102.03–2014.40 granted by Vietnam National Foundation for Science and Technology Development (Nafosted).

References

[1] Baar, Thomas, and Slavisa Markovic: *A graphical approach to prove the semantic preservation of UML/OCL refactoring rules.* Lecture Notes in Computer Science, Vol. 4, 2007, pp. 70-83.

[2] Berardi, Daniela and Calvanese, Diego and De Giacomo, Giuseppe: *Reasoning on UML class diagrams.* Journal of Artificial intelligence, Vol. 168, pp. 70–118, 2005, Elsevier.

[3] Bruegge, Bernd and Dutoit, Allen H: *Object-Oriented Software Engineering Using UML, Patterns, and Java.* 817 pages, Prentice Hall Press publisher, 2009.

[4] Cinnéide, Mel O and Nixon, Paddy: *Composite refactorings for Java programs.* In: Workshop on Formal Techniques for Java Programs, ECOOP, 2000.

[5] Evans, Andy S: *Reasoning with UML Class Diagrams.* Workshop on Industrial Strength Formal Method, IEEE Press, 1998, pp. 102–113.

[6] Folli, Alessandro, and Tom Mens: *Refactoring of UML models using AGG.* Electronic Communications of the EASST, Vol. 8, 2008.

[7] Fowler, Martin and Beck, Kent: *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[8] Gogolla, Martin and Richters, Mark: *Transformation rules for UML class diagrams.* Proc UML 98 Workshop, Springer-Verlag, Berlin, 1998, pp. 92-106.

[9] Hamilton, Kim and Miles, Russell: *Learning UML 2.0.* Vol. 286, 286 pages, O'Reilly publisher, 2006.

[10] Markovic, Slavisa: *Model refactoring using transformations*, PhD Thesis, 159 pages, EPFL publisher, 2008.

[11] Massoni, Tiago: *An approach to invariant-based program refactoring.* Electronic Communications of the EASST, Vol. 3, 2007.

[12] Mens, Tom and Demeyer, Serge and Janssens, Dirk: *Formalising behaviour preserving program transformations.* International Conference on Graph Transformation, 2002, pp. 286-301, Springer.

[13] Mens, Tom and Taentzer, Gabriele and Runge, Olga: *Analysing refactoring dependencies using graph transformation.* Journal of Software & Systems Modeling, Vol. 6, No. 3, 2007, pp. 269–285, Springer.

[14] Najafi, Mehrnaz and Haghighi, Hassan and Zohdi Nasab, Tahereh: *A Set of Refactoring Rules for UML-B Specifications.* Computing and Informatics, Vol. 35, 2016, No. 2, pp. 411-440.

[15] Nikulchev, Evgeny and Deryugina, Olga: *Model and Criteria for the Automated Refactoring of the UML Class Diagrams*. International Journal of Advanced Computer Science and Applications, Vol. 7, No. 12, 2016, pp. 76–79.

[16] Opdyke, William F.: *Refactoring object-oriented frameworks*, University of Illinois at Urbana-Champaign, PhD thesis, 1992.

[17] Parkinson, Matthew: *Class Invariants: The end of the road?* Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO), 2007.

[18] Pereira, Claudia and Favre, Liliana and Martinez, Liliana: *Refactoring UML Class Diagram*. In: Proceedings of 2004 Information Resources Management Association International Conference (IRMA 2004).

[19] Poo, Danny and Kiong, Derek and Ashok, Ms Swarnalatha: *Object, Class, Message and Method*. In: Object-Oriented Programming and Java, 2008, pp. 7–15, Springer.

[20] Porres, Ivan: *Model refactorings as rule-based update transformations*. In: International Conference on the Unified Modeling Language. Springer Berlin Heidelberg, 2003, pp. 159-174.

[21] Sunyé, Gerson and Pollet, Damien and Le Traon, Yves and Jézéquel, Jean-Marc: *Refactoring UML models*. International Conference on the Unified Modeling Language, 2001, pp. 134–148, Springer.

[22] Wimmer, Manuel and Perez, Salvador Martínez and Jouault, Frédéric and Cabot, Jordi: *A Catalogue of Refactorings for Model-to-Model Transformations*. Journal of Object Technology, Vol. 11, No. 2, 2012.

[23] Whittle, Jon. *Transformations and software modeling languages: Automating transformations in UML*. Proc. of UML 2002 - The Unified Modeling Language. Lecture Notes in Computer Science, Springer, 2002, pp. 57–63.

[24] Van Gorp, Pieter and Stenten, Hans and Mens, Tom and Demeyer, Serge: *Formal UML Support for the semi-automatic Application of object-oriented Refactorings*. University of Antwerp, Technical Report, 2003.