Composition of UML Described Refactoring Rules[*]

Slavisa Markovic

Swiss Federal Institute of Technology
Department of Computer Science
Software Engineering Laboratory
1015 Lausanne-EPFL
Switzerland
e-mail: Slavisa.Markovic@epfl.ch

Abstract. Refactorings represent a powerful approach for improving the quality of software systems. A refactoring can be seen as a special kind of behavior preserving model transformation. The Object Constraint Language (OCL) together with the metamodel of Unified Modeling Language (UML) can be used for defining rules for refactoring UML models. This paper investigates descriptions of refactoring rules that can be checked, reused and composed. The main contribution of this paper is an algorithm to compute the description of sequentially composed transformations. This allows one to check if a sequence of transformations is successfully applicable for a given model before the transformations are executed on it. Furthermore, it facilitates the analysis of the effects of transformation chain and its usage in other compositions

1 Introduction

Model transformations are the core of Model Driven Architecture (MDA) approach [10]. There exist several classifications of model transformations like classification on “exomorphic”, “endomorphic” and “creational” transformations [12].

Exomorphic transformation is a special type of transformation that involves models from different levels of abstraction. On the contrary to exomorphic transformation, endomorphic transformation deals with models that are represented at the same level of abstraction and where source and target models are instances of the same metamodel. Usages of endomorphic transformations are numerous. Typical examples of this kind of transformation are refactorings [1, 2, 3].

For the description of model transformations we will use the Object Constraint Language [8] that would replace, possibly vague and ambiguous, natural language description. In this paper, we will address the problem of composing endomorphic model transformations in the context of their application to UML models [7]. Namely, the aim is to have a method that will allow us to specify a transformation, analyze that transformation, evaluate a sequence of transformations or compose together descriptions of different transformations. This will leave us with the possibility to decide in

[*] This paper has been supported by Swiss National Scientific Research Fund under the reference number 2000-067917.

advance if it is meaningful or safe to perform a certain sequence of transformations or not. We offer some examples in this paper that show the composition of several transformation descriptions, with the goal of creating more elaborated transformation descriptions.

This paper is composed as follows: section 2 contains details about the concepts used in this approach and gives some examples of transformation descriptions applied on refactoring rules. In section 3, the approach for composition of transformations is presented. Sections 4 and 5 are reserved for related work and conclusions, respectively.

2 Refactoring

Refactoring can be seen as a process of improving structure of a software system without changing its behavior. In other words it means that for the same input the refactored software system has to produce the same output as before. This behavior preservation is assured by so called “preconditions” and “postconditions”. Preconditions have to be fulfilled before some refactoring is executed, and postconditions represent description of effects of one refactoring In this paper we show that OCL constraints can be used to describe these pre and postconditions.

In the past, as described in [1], the process of refactoring was based on code-to-code transformation. Today, thanks to UML, we can raise the level of abstraction by refactoring UML models instead of implementation code. Because of this model change, which appears after applying refactoring, we can observe refactoring as just one type of model transformation. This type of model transformations, where the transformation will be executed only if the precondition is met, is called “Conditional Transformation” in [10]. Our main emphasis is on the pre and postconditions that have to be satisfied in order to refactor some UML model. Applying OCL assertions on instances of the UML metamodel (i.e., model elements) is possible to define these constraints. The two following sub-sections contain examples of OCL descriptions of transformations that represent two refactoring rules. The first one, “Abstraction”, creates a new abstract superclass for some existing class. The second one creates a new interface for some existing class.

2.1 Description of the “Abstraction” Transformation

This transformation is used to create a new abstract class that will be connected with a generalization link to the existing class, e.g. it creates a new parent of the existing class.

The precondition of the description assures that before executing the transformation there is a class in the system with the same name as the parameter *product* but there is no class with the same name as the parameter *absProduct*.

The postcondition of the description assures that after execution of the transformation a new abstract class with the same name as the parameter *absProduct* is created and that there exists a generalization link between the new and initial class.

The postcondition is based on frame assumption that means that anything that is not mentioned has not been changed during the transformation.

Example of states that represent this refactoring are shown on figure 1. The upper and lower parts of figure represent the same system state but using different syntax. The upper part describes one system state using UML model elements, and the lower one describes the same system state using instances of metemodel elements. The OCL constraints are specified on the metamodel level. The relevant part of the UML metamodel is shown on figure 2.

```
context Package
def: classes: Set(Class) =
self.ownedClassifier->select(oclIsTypeOf(Class))->collect(c|c.oclAsType(Class))
def: generalizations: Set(Generalization)=
self.ownedClassifier->collect(generalization)
def: interfaces: Set(Interfaces::Interface)=
self.ownedClassifier->select(oclIsTypeOf(Interfaces::Interface))
     ->collect(c|c.oclAsType(Interfaces::Interface))

context Package::abstraction (product:String, absProduct:String)
pre:
     classes->exists(name= product) and
     not classes->exists(name= absProduct)
     not interfaces->exists(name= absProduct)
post:
let: absProd:Class=classes->select(name=absProduct)->any(true) in
let: gen:Generalization=generalizations->select(g|g.specific.name= product
          and g.general.name= absProduct)->any(true) in
     absProd.isAbstract=true and
     absProd.oclIsNew() and
     gen.oclIsNew()
```

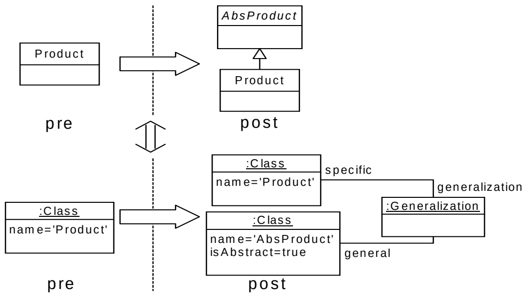

Fig. 1. Example of the “Abstraction” refactoring

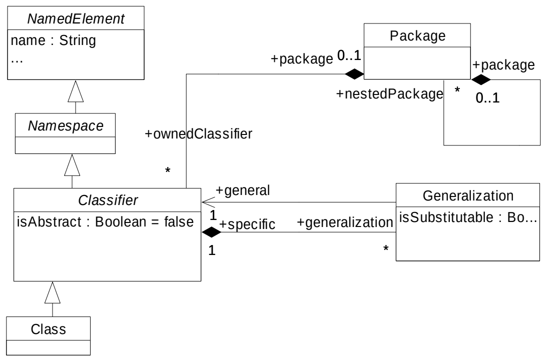

Fig. 2. The relevant part of the UML 2.0 metamodel for the "Abstraction" refactoring

2.2 Description of the "Interface Extraction" Transformation

The Interface Extraction transformation is used to add an interface to a class. This enables another class to take a more abstract view of a class by accessing it instead, via an interface.
By the pre condition of the description we assert that there exists a class on which we want to apply this transformation and that there does not exist neither a class nor an interface with name *creatorInf*.

The postcondition of the description ensures that, after execution of the transformation, a new interface is created with the same name as *creatorInf* parameter, and that there exists one implementation relation between this interface and the class whose name we pass as a parameter "creator". Furthermore, the postcondition ensures that all public operations from the class must exist in the interface. In this expression we have "borrowed" *hasSameSignature* from UML 1.5 [15] that does not exist in the UML 2.0.

This transformation is described in the following way using OCL constraints applied on the instances of the UML metamodel.

```
context Package
def: implementations:Set(Interfaces::Implementation)=
self.ownedClassifier->collect(implementation)

context Package::interfaceExtraction (creator:String, creatorInf:String)
pre:
    classes->exists(name= creator) and
    not classes-> exists(name= creatorInf) and
    not interfaces-> exists(name= creatorInf)
```

```
post:
let: creatInf:Interface=interfaces->select(name=creatorInf)->any(true) in
let: imp:Implementation=implemenations->
            select(i:Implementation|i.ilementatingClassifier.name= creator and
                                        i.contract.name= creatorInf)->any(true) in
let: creat:Class=classes->select(name=creator)->any(true) in

    creatInf.oclIsNew() and
    imp.oclIsNew and
    creat->collect(operation)->select(visibility=VisibilityKind::public)->
      forAll(o1:Operation|creatInf->collect(operation)->
        exists(o2:Operation|o2.hasSameSignature(o1))) and
    creatInf->collect(operation)->forAll(c:Operation| c.oclIsNew())
```

Figure 3 shows one possible state that satisfies the postcondition of the “Interface Extraction” transformation.

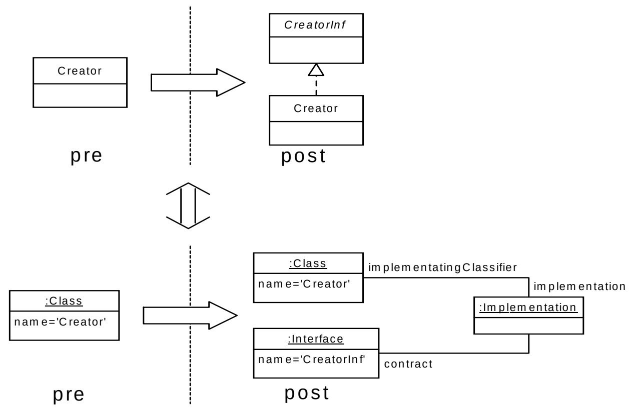

Fig. 3. Example of “Interface Extraction” refactoring

Figure 4 shows the relevant part of UML 2.0 metamodel on which the OCL expressions from pre and postconditions of “Interface Extraction” are specified.

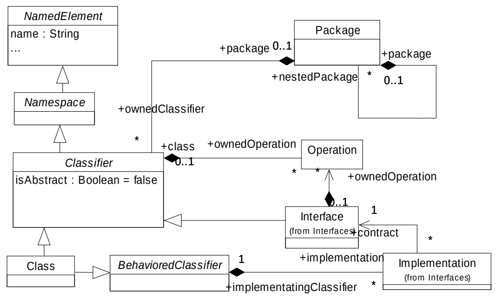

Fig. 4. The relevant part of the UML 2.0 metamodel for the “Abstraction” refactoring

3 Composition of Refactorings

3.1 Why Composing Refactorings?

Refactorings as they are presented in well known catalogs like [1] or [3] represent only “basic” transformations that can only be used in some specific situations. Furthermore, tools like [14] that support these “basic” refactorings do not let users to create their own refactorings, from scratch or by composing existing refactorings into new ones. The composition of refactorings would allow users of the tool to create their own complex refactorings that fulfill their specific needs. Of course, the new refactoring created in this way could be combined with others in order to create new ones, and so on.

The problem with composing refactorings that are given as the transformations of existing code or models is that if it is not possible to execute one component transformation we have to invoke a kind of roll-back mechanism that will restore the system in the previous state; it may even be necessary to rollback several steps. By not dealing with concrete transformations but only with their specifications, it is possible to avoid this kind of problem. Namely, the preconditions and postconditions of one composed transformation could be calculated before the transformation is executed so it would be possible to know in advance if one composition of transformations is legal and if it fulfills our needs. Furthermore, it is easier to analyze composed refactorings if they are represented by one pre-post pair.

3.2 Principles of Composing

Whenever we want to make a transformation that is composed of several transformations we have to deal with as many pre and post conditions as we have transformations.

The question that arises is how to calculate the “overall” pre and post conditions that we can use for the whole chain of small transformations.

Figure 5 shows one transformation that consists of three sub-transformations. As we have already stated each transformation has its own pre and post condition given by two OCL expressions. One approach for applying this sequence is to evaluate the pre and post expressions for each step (i.e., each transformation) in the given sequence on their own system states. A better approach would be to calculate one pre and one post condition for the whole sequence. This problem is solved by calculating pre and post condition for a chain of length 2. Iterative additions of the next transformation pre and post conditions will lead us to composite conditions for the chain of any length.

At a first glance, one might guess that two transformations can be composed simply by composing the preconditions using the AND operator and composing the postconditions using AND as well. In some cases this is correct but this would not always lead to the legal chain of transformation descriptions. This is because each of the conditions is evaluated on its specific system state in some point of time.

Our task now is to "project" precondition of intermediate state to condition of the initial state. Here, the semantics of the first transformation has to be taken into account. Obviously, if the precondition of the intermediate state is a logical consequence from the postcondition of the first transformation then nothing has to be added to the precondition of the initial state.

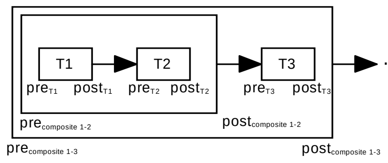

Fig. 5. Composite transformation

Figure 6 shows relation between two transformations. Each precondition is evaluated on its own system state that can be different from other system states. Also, each postcondition is evaluated on the separate system state but on the contrary to preconditions, each postcondition can be dependent on some part of the previous state. This dependency on the previous system state is realized using *@pre* operator in OCL. On figure 6, this kind of dependency is represented with the dashed lines.

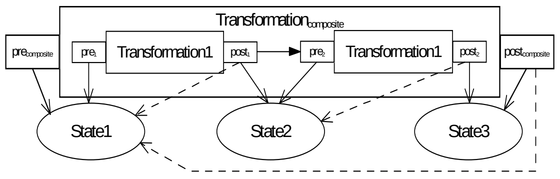

Fig. 6. Dependencies between two transformations

Before we proceed with the composition of transformation descriptions we have to perform a kind of "pre-processing" of descriptions by marking all variables dependant on their system states on the following way:

$$pre_i : V \to V_i$$
$$post_i : V \to V_{i+1}$$
$$post_i : V@pre \to V_i \quad (1)$$

In order to provide more precise description of transformation composition we introduce the following notation:

X – set of all variables that exist in some system state
X_1, X_2, X_3 – set of all variables in initial, intermediate and final system state
$pre_i(X_i) \mapsto \{true, false\}$ – precondition of the i-th transformation that is evaluated on the state X_i
$post_i(X_i, X_{i+1}) \mapsto \{true, false\}$ – postcondition of the i-th transformation that is evaluated on the states X_i and X_{i+1}
$trans_i(X_i, X_{i+1})$ – description of a transformation that transforms one system state i to system state i+1
$PRE(X_1) \mapsto \{true, false\}$ – composite precondition that is evaluated on initial system state
$POST(X_1, X_3) \mapsto \{true, false\}$ – composite postcondition that is evaluated on the states X_1 and X_3
$TRANS(X_1, X_3)$ – description of composite transformation that transforms initial system state to system state 3

Now we can represent each transformation description like:

$$trans_i(X_i, X_{i+1}) = pre_i(X_i) \wedge post_i(X_i, X_{i+1}) \quad (2)$$

Using this notation, two transformations from figure 6 can be represented as following:

$$trans_1(X_1, X_2) = pre_1(X_1) \wedge post_1(X_1, X_2) \quad (3)$$
$$trans_2(X_2, X_3) = pre_2(X_2) \wedge post_2(X_2, X_3)$$

The process of finding composed transformation description can be formulated like: calculating pre and postcondition for a transformation that transforms one system from state X_1 to state X_3. Given with our notation, composite transformation can be represented like:

$$TRANS(X_1, X_3) = PRE(X_1) \wedge POST(X_1, X_3) \quad (4)$$

Or in other words:

$$PRE(X_1) \wedge POST(X_1, X_3) \rightarrow \exists X_2 \; pre_1(X_1) \wedge post_1(X_1, X_2) \wedge pre_2(X_2) \wedge post_2(X_2, X_3) \qquad \textbf{(5)}$$

We can calculate composite precondition $PRE(X_1)$ using the following formula:

$$PRE(X_1) \rightarrow \exists X_2 \; pre_1(X_1) \wedge post_1(X_1, X_2) \wedge pre_2(X_2) \qquad \textbf{(6)}$$

On the similar way we can calculate the composite postcondition by using the following formula:

$$PRE(X_1) \wedge POST(X_1, X_3) \rightarrow pre_1(X_1) \wedge \exists X_2 \; post_1(X_1, X_2) \wedge post_2(X_2, X_3) \qquad \textbf{(7)}$$

The first condition for our chain of refactorings to be legal is that every precondition must evaluate to true on its own system state.

$$eval\ [pre_i(X_i)] = true \qquad \textbf{(8)}$$

If this condition is not satisfied then we have one illegal component transformation and we can conclude that our whole chain is illegal.

The second condition that must be satisfied is that if the postcondition of the first transformation description evaluates to true then the precondition of the second transformation description must also evaluate to true:

$$\text{if } eval\ [post_1(X_1, X_2)] = true \text{ then } eval\ [pre_2(X_2)] = true \qquad \textbf{(9)}$$

$$\Downarrow$$

$$eval[post_1(X_1, X_2)\ implies\ pre_2(X_2)] = true$$

Before we continue with composing OCL statements we are obliged to perform a sort of analysis. This is analysis of consistency between postcondition of the first and precondition of the second transformation description. With this step we detect all cases in which after completion of the first transformation we cannot continue with the second transformation due to its unsatisfied precondition.

During this analysis of legality of a chain of transformations three cases can be detected:

– The chain is legal
– The chain is illegal but it can be transformed into a legal one
– The chain is illegal but it can not be transformed into a legal one

The three following subsections contain three simple examples that describe these three possible cases.

Example 1. In this subsection we give an example of one legal chain of two transformations. These two transformations are shown in Table1.

Table 1. Example of two transformations before “pre-processing“

$trans_1$		$trans_2$	
pre_1	$post_1$	pre_2	$post_2$
$x > 2$	$x = x@\text{pre} + 2$	$x > 3$	$x = 5$
$y = 3$	$y = y@\text{pre}$		$y = y@\text{pre}$

After the “pre-processing” using (1) we have the following pre and postconditions like shown in Table 2.

Table 2. Example of one legal chain of two transformations

$trans_1$		$trans_2$	
pre_1	$post_1$	pre_2	$post_2$
$x_1 > 2$	$x_2 = x_1 + 2$	$x_2 > 3$	$x_3 = 5$
$y_1 = 3$	$y_2 = y_1$		$y_3 = y_2$

Now, by observing values of postcondition of $trans_1$ and precondition of $trans_2$ for variables of system state 2 we can conclude that sequence of transformations { $trans_1, trans_2$ } is legal because after solving the equation from the postcondition of $trans_1$, we have situation that $x_2 > 4$ in $post_1$ and $x_2 > 3$ in pre_2 which depicts a set of correct system states. In other words, any value that satisfies $post_1$ also satisfies pre_2, which means that the condition (2) is satisfied.

For example given in Table 2, using (6) we have calculated the composite precondition described with the following expression:

$$PRE\left(\{x_1, y_1\}\right) \rightarrow \exists x_2 y_2 \ x_1 > 2 \wedge x_2 = x_1 + 2 \wedge x_2 > 3 \wedge y_1 = 3 \wedge y_2 = y_1 \qquad \textbf{(10)}$$

After the first step of simplification we get:

$$PRE\left(\{x_1, y_1\}\right) \rightarrow x_1 > 2 \wedge x_1 > 1 \wedge y_1 = 3 \qquad \textbf{(11)}$$

And in the next step of simplification:

$$PRE\left(\{x_1, y_1\}\right) \rightarrow x_1 > 2 \wedge y_1 = 3 \qquad \textbf{(12)}$$

One remark is that during the process of calculating the composite precondition, we are always looking for the weakest solution among all preconditions that solve the condition (11).

Using formula (7) we can calculate the composite postcondition. For our example from Table 2 we have that:

$$PRE\left(\{x_1, y_1\}\right) \wedge POST\left(\{x_1, y_1, x_3, y_3\}\right) \rightarrow \tag{13}$$
$$\exists x_2 y_2 \;\; x_1 > 2 \wedge y_1 = 3 \wedge x_2 = x_1 + 2 \wedge y_2 = y \wedge x_3 = 5_1 \wedge y_3 = y_2$$

So after simplification we get:

$$POST\left(\{x_1, y_1, x_3, y_3\}\right) \rightarrow x_3 = 5 \wedge y_3 = 3 \tag{14}$$

Example 2. In this subsection we give an example of one illegal chain of two transformations that can be transformed into a legal one. Two transformations that build this chain are shown in Table 3. The only difference from example 1 is that now the precondition of the second transformation is $x_2 > 10$.

Table 3. Example of an illegal chain that can be transformed into a legal one

$trans_1$		$trans_2$	
pre_1	$post_1$	pre_2	$post_2$
$x_1 > 2$	$x_2 = x_1 + 2$	$x_2 > 10$	$x_3 = 5$
$y_1 = 3$	$y_2 = y_1$		$y_3 = y_2$

By observing values of postcondition of $trans_1$ and precondition of $trans_2$ for variables of system state 2 we can conclude that this sequence of transformations is not legal because after solving the equation from the postcondition of $trans_1$, we have situation that $x_2 > 4$ in $post_1$ and $x_2 > 10$ in pre_2. This means that there can exist system states that satisfy the postcondition of the first transformation but don't satisfy the precondition of the second one, i.e. there can exist some system states that do not satisfy (2).

If we want to compute composite precondition for example from Table 3 we follow the same algorithm like we did for example from Table 2. After the first step of simplification we have the following situation:

$$PRE\left(\{x_1, y_1\}\right) \rightarrow x_1 > 2 \wedge x_1 > 8 \wedge y_1 = 3 \tag{15}$$

Further simplification of this expression yields the following:

$$PRE\left(\{x_1, y_1\}\right) \rightarrow x_1 > 8 \wedge y_1 = 3 \tag{16}$$

This means that if we want to have legal composite transformation description we have to make composite precondition "stronger" than precondition of the first transformation description.

The composite postcondition will be the same as (14) because the change on the precondition of the second transformation does not make any influence on it.

Example 3. In this subsection we give an example of one illegal chain of two transformations that can not be transformed into a legal one. Two transformations that build this chain are shown in Table 4. The only difference from example 1 is that now the precondition of the second transformation is $x_2 < 4$.

Table 4. Example of an illegal chain that can not be transformed into a legal one

$trans_1$		$trans_2$	
pre_1	$post_1$	pre_2	$post_2$
$x_1 > 2$	$x_2 = x_1 + 2$	$x_2 < 4$	$x_3 = 5$
$y_1 = 3$	$y_2 = y_1$		$y_3 = y_2$

By observing values of postcondition of $trans_1$ and precondition of $trans_2$ for variables of system state 2 we can conclude that this sequence of transformations is not legal because after solving the equation from the postcondition of $trans_1$, we have situation that $x_2 > 4$ in $post_1$ and $x_2 < 4$ in pre_2. This means that set of system states that satisfies both pre and postcondition is empty.

If we try to compute the composite precondition using (7) we will get the following:

$$PRE(\{x_1, y_1\}) \rightarrow x_1 < 4 \wedge x_1 > 4 \wedge y_1 = 3 \qquad \textbf{(17)}$$

This means that it is not possible to find the composite precondition for this chain, i.e. the chain is illegal.

3.3 Example of a Composed Refactoring Rule

By analyzing our two refactoring examples from section 2, we can conclude that composed transformation description will look like the following:

```
context Package::composed (product:String, absProduct:String, creator:String,
creatorInf:String)
pre:
     classes->exists(name= product) and
     not classes->exists(name= absProduct) and
     not interfaces-> exists(name= absProduct)
     classes->exists(name= creator) and
     not classes-> exists(name= creatorInf) and
     not interfaces-> exists(name= creatorInf) and
     not absProduct=creatorInf
post:
let: absProd:Class=classes->select(name=absProduct)->any(true) in
let: gen:Generalization=generalizations->select(g|g.specific.name= product
          and g.general.name= absProduct)->any(true) in
```

```
let: creatInf:Interface=interfaces->select(name=creatorInf)->any(true) in
let: imp:Implementation=implemenations->
             select(i:Implementation|i.ilementatingClassifier.name= creator and
                                     i.contract.name= creatorInf)->any(true) in
let: creat:Class=classes->select(name=creator)->any(true) in
     absProd.isAbstract=true and
     absProd.oclIsNew() and
     gen.oclIsNew() and
     creatInf.oclIsNew() and
     imp.oclIsNew and
     creat->collect(operation)->select(visibility=VisibilityKind::public)->
       forAll(o1:Operation|creatInf->collect(operation)->
         exists(o2:Operation|o2.hasSameSignature(o1))) and
     creatInf->collect(operation)->forAll(c:Operation| c.oclIsNew())
```

3.4 Limitations of the Approach

One can notice that in our approach we use only a limited set of OCL expressions in postconditions in order to be able to evaluate unique new system state from the initial one. This means that we do not use expressions with, for the example "<" or ">" operators. In other words, we are using only those OCL expressions that can be evaluated to true only on one system state. Using the full set of OCL expressions can be seen as superfluous because the future system state in which we want to bring our system is precisely defined so operations that can evaluate to true on several system states have to be avoided. Also, OCL is clumsy when we do not use the frame assumption but have to write full versions of pre or postconditions. Still there is a hope that this process of extracting full version of OCL constraints out of version that is based on the frame assumption could be automated.

Current limitation of our approach is that we can describe only transformations whose source and target models are instances of the same metamodel like the refactoring rules.

4 Related Work

In his PhD thesis [1], Opdyke represents refactorings as combinations of preconditions whose purpose is the preservation of behavior of a program that is to be refactored and actual refactoring steps that are described using natural language. These preconditions have to preserve seven basic properties of each program that are usually violated during refactoring process. Also, this work contains certain principles of composition of refactorings. The emphasis is on the composition of primitive preconditions in order to keep behavior preserved.

In [2], Roberts introduces postconditions into the refactoring process; they are used in the composition of preconditions for the chain of refactorings. Before composing preconditions, Roberts first transforms them according to the previous refactorings in the chain, by using the postconditions. Besides the problem of composition, he discusses the problems of commutativity and dependencies between refactorings. He extends the approach of Opdyke by formalizing refactorings using first-order predicate logic. Similarly to this approach, we also use postconditions in order to calculate cumulative precondition of a refactoring chain.

In his thesis [5], Ó Cinneidé uses the approach of Roberts in order to compose refactorings related to design patterns. Besides the composition of preconditions, he introduces the concept of composition of postconditions. He uses this approach for manually calculating design patterns seen as a chain of primitive refactorings.

Sunyé et al. [4] use a very similar approach to our one for representing refactorings. They describe refactorings using OCL expressions applied to the metamodel of UML. A difference of our approach with theirs is that we localize all transformations on the level of one package assuming that, for the sake of simplicity, all refactorings are done locally in the package. Their refactorings are defined on the model elements that are influenced by the specific refactoring. In [13] , Sunyé et al. extend OCL with so called "actions" which are used to describe the transformations as the third component of refactoring, besides pre and postconditions. Composition of refactorings is not discussed in this work.

Due to the lack of Action Semantics in UML 1.4 specification, Van Gorp et al. [6] propose their own UML extension for the support of refactorings. OCL expressions are applied on the UML 1.4 extended metamodel. They define OCL pre and post conditions as the separate functions introduced with the OCL def-let mechanism. This approach lacks of the possibility to use the @pre operator inside of postconditions.

Kniesel and Koch [11] give a precise definition of refactorings composition and describe an automatic approach of composing refactorings that are given with their precondition and transformation description. Their notion of transformation description is different from ours, because they use this term as a function that transforms conditions to reflect the "real" transformations. Our transformation descriptions include necessary pre and postconditions which describe each transformation. Their approach for composition of refactorings is different to ours because they start composing from the last transformation while we start from the first one.

5 Conclusion

This work tackles the problems of composing endomorphic model transformations like refactorings. The aim was to provide an approach for validating the sequence of these transformations and to find the way of composing these transformations in order to get more complicated transformations that can be applied or used in further compositions. On the contrary to work of some authors where a new metamodel is created in order to facilitate usage of OCL expressions, we use the full UML. Despite all its drawbacks, OCL in this approach provides an elegant way of describing model transformations and compositions of transformations. Unfortunately, by using OCL pre-post conditions like in our approach we can deal only with the models that represent instances of the same metamodel. Our wish is to further develop our method in order to be able to deal not only with endomorphic transformations but with any kind of transformation.

The future steps will involve the creation of a library of mini transformation descriptions in order to get some foundations for describing more complicated refactorings or design patterns. Our intention is to create the tool that would be able to automatically calculate dependencies and mismatching between mini-transformations descriptions and to be able to generate a composite transformation description out of these mini-transformation descriptions.

References

[1] Opdyke, W. F., “Refactoring Object-Oriented Frameworks,” Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[2] Roberts, D., “Practical Analysis for Refactoring,” Ph.D. thesis, University of Illinois at Urbana-Champaign, 1999.

[3] Fowler, M., “Refactoring: Improving the Design of Existing Programs,” Addison-Wesley, 1999.

[4] Sunyé, G., Pollet, D., Le Traon Y., and Jézéquel J-M. “Refactoring UML models,” In Gogolla M. and Kobryn C., editors, Proc. UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, volume 2185 of LNCS, pages 134–148. Springer, 2001.

[5] Ó Cinnéide, M., “Automated Application of Design Patterns: A Refactoring Approach,” Ph.D. thesis, Department of Computer Science, Trinity College, University of Dublin, 2000.

[6] Van Gorp, P., Stenten, H., Mens T., Demeyer, S.: “Towards automating source-consistent UML Refactorings,” UML 2003.

[7] Object Management Group. UML 2.0 Superstructure Specification, September 2003

[8] Object Management Group. UML 2.0 OCL Specification, Final Adopted Specification, October 2003

[9] Gamma, E., Helm R., Johnson R. and Vlissides, J. “Design Patterns: Elements of Reusable Object-Oriented Languages and Systems,” Addison-Wesley, 1994.

[10] Object Management Group. MDA Guide Version 1.0.1, OMG, 1. June 2003

[11] Kniesel G., Koch H., “Static Composition of Refactorings”, In R. Lämmel, editor, Science in Computer Programming; Special issue on program transformation. Elsevier Science, 2004. To appear.

[12] Pollet D., Vojtisek D. and Jézéquel J-M. “OCL as a core UML transformation language.” WITUML 2002 Position paper, Malaga, Spain, June 2002.

[13] Sunyé, G., Le Guennec, A. and Jézéquel J-M. “Using UML Action Semantics for model execution and transformation.” Information Systems Journal 27(6):445-457, July 2002

[14] Eclipse Project, http://www.eclipse.org/

[15] Object Management Group. Unified Modeling Language (UML), version 1.5, March 2003.