Model Transformation Contracts and their Definition in UML and OCL

Eric Cariou, Raphaël Marvie, Lionel Seinturier and Laurence Duchien

Laboratoire d'Informatique Fondamentale de Lille (LIFL)
Université des Sciences et Technologies de Lille
UMR CNRS 8022 - INRIA Futurs
59655 Villeneuve d'Ascq Cedex - France
{cariou, marvie, seinturi, duchien}@lifl.fr

April 2004

Abstract. A major challenge of the OMG Model-Driven Architecture (MDA) initiative is to be able to define and execute transformations of models. Such transformations may be defined in several ways and with various motivations. Our motivation is to specify model transformations independently of any transformation technology. To achieve this goal, we propose to define transformation contracts. We argue that model transformation contracts are an essential basis for the MDA, they can be used for specification, validation and test of transformations. This paper focuses on model transformation contract specification. We investigate the way to define them using standard UML and OCL features. In addition to presenting the approach and some experimental results, this paper discusses some limits of standard OCL to define transformation contracts as well as some hints to bypass these limitations.

---

1 Introduction

The Model-Driven Architecture (MDA) initiative of the OMG has made its way in the software community. Its main goal is to shift the software development problematic from technical issues to abstract specification of an application. The MDA defines a process in which Platform Independent Models (PIM) are central. They set the focus on the business parts of an application, independently of any technical or architectural target. PIM level models are refined and transformed in order to define PSM (Platform Specific Models) level models, according to technical choices.

Then, one major challenge of the MDA is to be able to define and execute transformations of models. In the past few years, both academic and industrial researchers have devoted a lot of efforts to study model transformation techniques. As a result, numerous proposals have emerged. In April 2002, the OMG issued a Request For Proposal on Query/Views/Transformations (QVT) [7]. It promises to deliver a standardized support for model transformation. However, the proposal is not finalized yet as several responses have been submitted without an agreement on their merging.

In addition, it is important to specify the pre-requisite as well as the expected output of a

model transformation. If MDA encourages the transformation of models, then first they can be chained and second one can expect to find off the shelf transformations. *Design by contract* [2, 10] has been accepted in the software community as a foundation for building trusted software components and applications. We propose to take benefits from this approach in the context of model transformations and to define *model transformation contracts*. Transformation contracts are defined using three sets of constraints: constraints on source model, constraints on target model and constraints on relationships among source and target model elements.

We argue that model transformation contracts are an essential basis for the MDA and more generally for Model-Driven Engineering (MDE). Indeed, they can improve processes made up of model transformations in several ways:

Specification and documentation A transformation contract allows a designer to specify what a transformation does, under which conditions it can be applied and what its excepted result is. These informations are also useful for choosing and applying the proper transformation in a given context.

Validation and test With the help of adequate tools, it will be possible to check if a model can be transformed by a given transformation or if a model is a valid result of this transformation. Transformation contracts can also be used as basis for testing models and transformations; they can notably serve for defining oracles.

Composition and chaining of transformations The MDA implies to execute a chain of transformations to go from PIM level to the code through the PSM level. Transformation contracts can help in checking if two or more transformations can be executed in sequence. Notably, for two chained transformations, constraints on source model of the second one must be compatible with constraints on target model of the first one. Moreover, contracts can serve to check or test the validity of the composition of several transformations into a more global one.

In this paper, we focus on the specification of transformation contracts. In the meantime, we are also working on transformation composition [9].

The rest of this paper is organized as follows. Section 2 gives a general definition of model transformation contracts and investigates the way to express them by using regular UML and OCL features. It also discusses some OCL limitations. Section 3 gives two examples of model transformation contract. Section 4 proposes OCL extensions to improve and facilitate contract transformation specification. Section 5 discusses related works, before concluding with some future trends.

2 Model Transformation Contracts

2.1 Specifying Transformation Contracts

The goal of a contract is to define the *what* and not the *how* [2, 10] of a piece of software. Applied to model transformations, a contract deals with defining what the transformation is expected to do, what the constraints for its use are, but without entering the details of how the transformation is performed.

On a general basis, a model transformation is applied to a source model and produces a target model. Such a transformation is performed using a transformation operation. Then,

a transformation contract should specify these three elements. We define a transformation contract as a tuple of three sets of constraints:

- a set of constraints to be matched for a model to be candidate as a source model of the transformation,
- a set of constraints to be matched for a model to be considered as a valid target model produced by the transformation, and
- a set of constraints on the evolution of elements from the source to the target model

The last set of constraints is generally defined under the form of pre- and post-conditions attached to the transformation operation.

2.2 Contract Definition Using UML and OCL

We have chosen to define model transformation contracts using standard modeling or specification languages only. Our motivation was not to define a new language for this purpose. We first investigate the use of standard UML [14] and OCL [17] for defining transformation contracts. Our first experimentations have been restricted to the transformation of UML class diagrams. However, this approach is to be applied to other UML diagrams such as sequence or state ones and to MOF models.

Defining constraints on source and target models is achieved using metamodeling techniques. Constraints on UML diagrams are expressed at the level of the UML metamodel. Adding constraints on a class diagram is done by defining constraints on the metamodel part specifying the structure and elements of class diagrams. So, constraints on source and target models can be expressed by constraining or specializing the UML metamodel. This can be done by defining a profile as proposed in the UML specification [14]. Such a profile defines concepts specific to a given modeling context and relationships among these concepts and standard UML features. In order to precisely define constraints attached to profile elements, invariants written in OCL have to be specified.

For expressing constraints dealing with relationships between the source and target model elements, we propose to use OCL and more precisely to define the transformation operation through pre- and post-conditions. The precondition statement deals with the state of the model before the transformation, *i.e.* the source model, and the postcondition with the state of the model after the transformation, *i.e.* the target model. The OCL `@pre` construction enables diagram element before the transformation operation call to be referenced in the postcondition specification. It allows then source model elements to be handled in the postcondition. So, in the postcondition, both source and target models can be referenced which enables evolution among both models' elements to be defined. Specifying the transformation operation in OCL is more complex than defining constraints on source and target metamodels. We discuss this OCL operation specification further on in the following section.

2.3 Defining Transformation Operation in OCL

When defining the transformation operation in OCL with pre- and post-conditions, we have to take into account two OCL characteristics:

1. OCL expressions must be defined in the context of a UML classifier [17, 13] (a class, an interface or a datatype on a class diagram). An operation specified in OCL is owned by a classifier.

2. OCL expressions are attached to a classifier and express constraints on this classifier, its associated classifiers and their relationships. These constraints have to be respected by instances of these classifiers[1]. An operation pre-condition defines constraints that must be respected by instances of these classifiers before the operation call and the post-condition defines constraints that must be respected by instances of the same classifiers after the operation call.

We want to define in OCL a model transformation operation for specifying constraints on the evolution of model elements between a source model and a target model. In this context, instances we talked above are these model elements. Model elements are instances of metamodel elements by principle. So, in order to respect both OCL characteristics we described, the operation transformation must be owned by a classifier of a metamodel (the UML metamodel or one of its specialization defined by a profile). A major constraint is also that source model and target model must be instances of the same metamodel.

Some may argue that talking about a relation of instantiation between a model and a metamodel is not correct. A model element is an instance of a metamodel element but this relationship does not really fit at the whole model level. It is then more precise to talk about a conformity relationship: a model conforms to its metamodel.

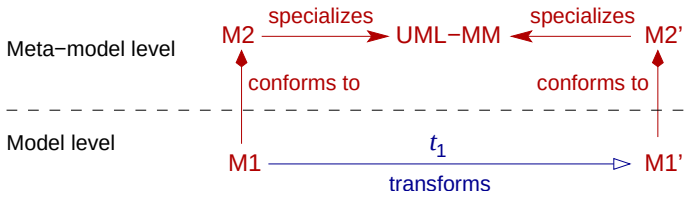

Figure 1: General relations between models, metamodels and the transformation operation

Figure 1 shows general relationships among models, metamodels and the UML metamodel. A transformation *t1* transforms a source UML model *M1* into a target UML model *M1'*. *M2* and *M2'* are profiles defining respectively constraints on source models and constraints on target models, *i.e.* they are source and target metamodels, and are both specializations of the UML metamodel (*UML-MM*). A model has to conform to its metamodel. If no particular constraints are required on a given model, its metamodel (*M2* and *M2'*) is the UML metamodel.

Before defining a transformation operation, we must then find the classifier of the proper metamodel that owns this operation. A proper metamodel is a metamodel for which both source and target models are conforming to. A major problem is often to find this metamodel.

We have identified three main contexts of relationships among models, specialized metamodels and the UML meta-model. Other cases exist but they can be adapted from these three cases that are the most common. They are represented on figure 2. Here are the details of the three cases:

(a) both source and target models are regular UML diagrams, no meta-level specialisation is required. The transformation operation is then defined in the context of a classifier of the UML metamodel.

[1]An interface is not instantiable. Constraints attached to an interface must be respected by instances (of classes) realizing this interface.

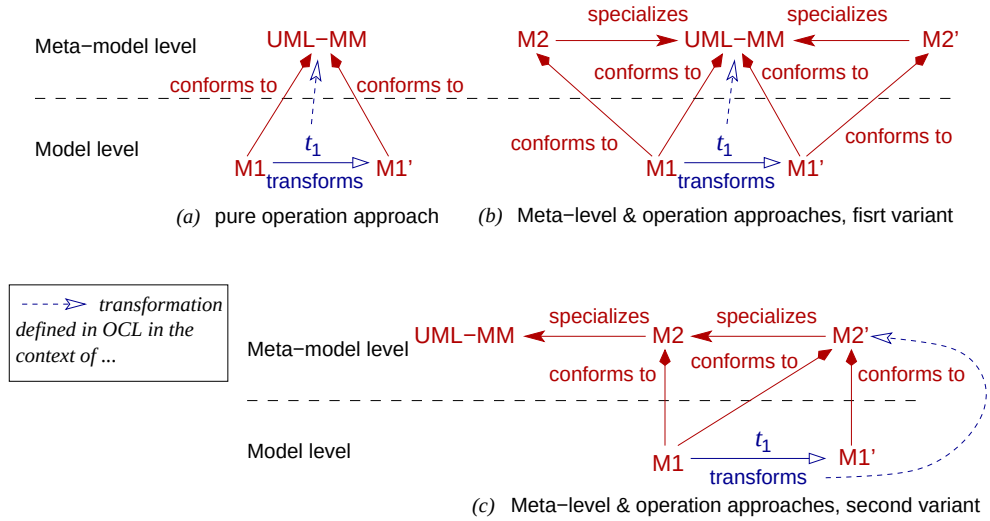

Figure 2: Three majors kinds of relations between models, metamodels and the transformation operation

(b) the source and the target metamodels are specialisations of the UML-metamodel. In this case, the transformation operation is defined in the context of a the UML-metamodel.

(c) the source metamodel is a specialisation of the UML metamodel and the target metamodel is a specialisation of the source one. In this case, the transformation is defined in the context of a the target metamodel (because both source and target models conform to this metamodel[2]).

The case *(a)* corresponds to the `addAccessors` transformation we define in section 3.2 on page 7. The case *(c)* to the `proxyAddition` transformation we define in section 3.3 on page 10 (with taking into account the problem we describe in next paragraph).

However, sometimes, it is not possible to find so easily a metamodel for which both source and target models conform to. Indeed, there can be contradictions between constraints added on each metamodel. For instance, the next section (3.3 on page 10) details a transformation in which a source metamodel implies that a Client class must have associations with a Server class. On the target metamodel side, a Client class must not have any association with a Server class. Thus, these two metamodels contain incompatible constraints. In this case, we can neither use the UML metamodel as context for the operation transformation because the concepts of Client and Server are not defined in it. The solution is then to define a new metamodel for which both source and target models are conforming to and that is sufficiently specialized (i.e. that defines the necessary concepts) to define the transformation operation specification.

Figure 3 defines the solution we propose for the example of the next section. *M2s* and *M2s'* are respectively the complete source and target metamodels. *M2'* is the target metamodel without the OCL constraints that will prevent the source model *M1* to conform to *M2'*. Then, *M2s'* is a specialization of *M2'* (*M2s'* is a specialization of *M2* by only adding OCL constraints, *M1'* then conforms to both *M2'* and *M2s'*).

[2]This property depends of course of the specialization relationships between *M2* and *M2'*. We are here in a context where this specialization is such that both source and target models conform to the target metamodel.

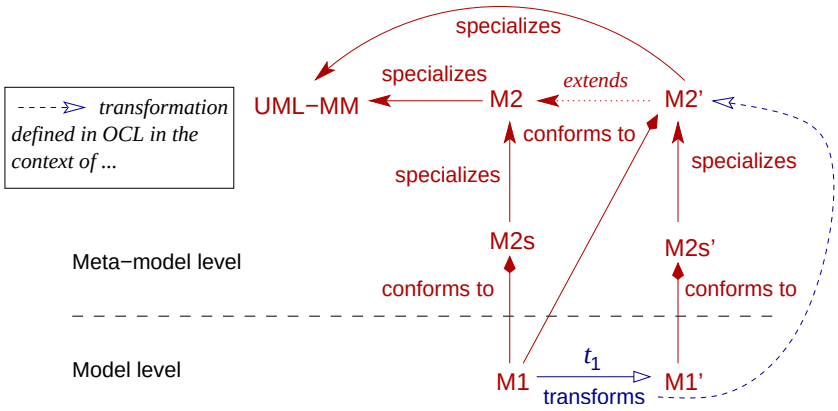

Figure 3: Detailed relations between models, metamodels and the transformation operation for case *(c)*

Once the common metamodel has been found, we need to find the classifier that will own the transformation operation. The only rule to follow is to choose a classifier allowing all necessary elements implied in the OCL expressions to be referenced through navigation on the metamodel class diagram. Most of the time, a lot of classifiers will verify this rule. In this case, one can choose the classifer that simplifies the writing of the OCL expressions.

2.4 Summary on Transformation Contract Specification in UML and OCL

We propose to define a transformation contract using UML and OCL with the three following elements:

- A profile defining the constraints on the UML source model
- A profile defining the constraints on the UML target model
- The OCL specification of the transformation operation. This operation must be attached to a classifier of a metamodel for which both source and target models conform to.

For the last point, the difficulty is sometimes to find this common metamodel.

3 Experimentation

In this section, we describe two examples of the definition of a transformation contract. To begin with, we present our simplified UML metamodel. Its goal is only to simplify the writing of OCL expressions at the metamodel level.

3.1 A Simplified UML Metamodel

The UML 2.0 metamodel [14] is quite complex. In particular, the complete class diagram specifying the UML is very large. As discussed in previous section, transformation contracts are expressed at the metamodel level, *i.e.* in the context of the UML metamodel. In order to ease experimentation and understanding, a simplified UML metamodel enough for experimenting transformations discussed in this paper has been defined. In addition to removing all

the non-useful elements of the UML metamodel, some elements and relationships have been modified slightly. Nevertheless, our examples defined in standard UML are usable with this lightweight version.

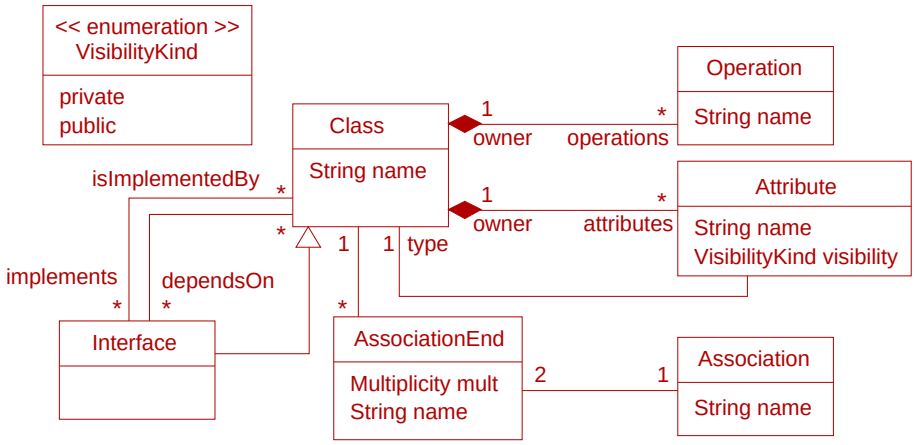

Figure 4: Our simplified UML metamodel

The simplified UML metamodel is defined by the class diagram depicted by figure 4. In order to be complete, the semantic of this metamodel elements should be given. Moreover, the well-formedness rules should be specified. As an example, the following OCL invariant states that an interface has no attribute:

context Interface **inv**: self.attributes –> isEmpty()

3.2 The "hello world" of transformations

Dealing with MDA transformations, a basic example commonly used is the privatization of classes' public attributes, that can be seen as the *hello world* of model transformations. Any attribute of a class which visibility is public introduces changes to the class definition. First, the visibility of such attribute is set to private. Second, two accessor methods are added to the class: one for getting the attribute value, named like the attribute prefixed with *get*, and one for setting the attribute value, named like the attribute prefixed with *set*.

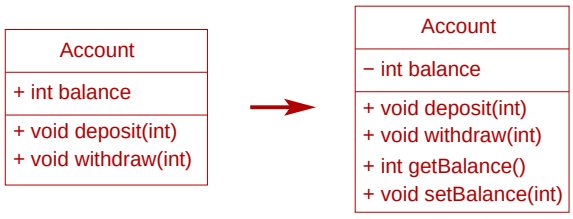

Figure 5: Example of the `addAccessors` transformation

Figure 5 depicts an example of such a transformation. On the left, a class defining a bank account is specified with a public attribute named balance. Applying the mentioned transformation to this class results in the class on the right. In this second version, visibility of

the balance attribute is set to private. Moreover, two accessors methods have been introduced to get (`getBalance`) and set (`setBalance`) the attribute value.

The transformation presented in this section is named `addAccessors` and its aim is to transform public attributes to private ones with associated getter and setter methods. More precisely, this transformation ensures that attributes of a model follow the rule of encapsulation (basis of the object approach)[3]. After applying the transformation, all defined attributes still exist but none of them have the public visibility. Moreover, a getter and a setter methods must be associated with each attribute.

3.2.1 Metamodel specializations

As explained above, we have to define two metamodels – a source and a target one – as specializations of the UML metamodel for expressing constraints on the metamodels according to the transformation.

In the context of the `addAccessor` transformation, the source metamodel is the UML metamodel (more precisely, the simplified version discussed in section 3.1), because no particular constraints are required on source models. The transformation can be performed on any kind of UML class diagrams. The target metamodel is a specialization based upon the definition of constraints that apply on attributes and method definitions of its model instances. These constraints are defined as a class invariant expressed in OCL. They have to be respected by all UML models that result from the `addAccessors` transformation. In other contexts, the metamodel class diagram structuration has to be modified as constraints may apply on the relation between classes.

```
1  context Class inv:
2  attributes -> forAll ( a : Attribute |
3      a.visibility = #private and
4      a.owner.operation -> exists (op : Operation |
           op.name = "get" + a.name) and
5      a.owner.operation -> exists (op : Operation |
           op.name = "set" + a.name))
```

Figure 6: Addition of OCL constraints on the UML metamodel for defining the target metamodel

Figure 6 depicts the invariant added on the UML metamodel to define the target metamodel. It states that each class attribute (line 2) has to respect the following constraints:

- the attribute visibility is private (another solution to express this constraint would be to remove the *public* item of the *VisibilityKind* enumeration class of the simplified UML metamodel –see figure 4) (line 3),
- a method is named like the attribute prefixed by *get*, the getter[4] (line 4), and
- a method is named like the attribute prefixed by *set*, the setter[4] (line 5).

[3]This transformation is not intended to be applied on a particular attribute as depicted in [15] but on whole class diagrams. This choice has been motivated by the ability of defining the same transformation simply by adding constraints on a metamodel. This aspect is further discussed later on.

3.2.2 Specification of the `addAccessors` operation

Before defining the operation in OCL with pre and post-conditions, we must find the proper classifier of the right metamodel that will own the transformation operation. The `addAccessors` operation transforms a regular class diagram into another call diagram with specific constraints. Models conforming to the target metamodel also conform to the UML metamodel. Indeed, the target metamodel defines that target class diagrams must have only private attribute with getter and setter methods. But, as no specific stereotypes are defined, target models conform also to the UML metamodel. So, the common metamodel is the UML metamodel. We are here in the case *(b)* of figure 2 on page 5 (with *M2* equals to the UML metamodel). For the classifier owning the operation, the best choice is the Class class. Any classfier of the UML metamodel can own the operation but defining it in the Class class context will simplify a lot the writing of the OCL expressions.

```
1  context Class:addAccessors()
2  pre: -- none
3  post:
4      attributes –> forAll ( a : Attribute | a.visibility = #private) and
5      attributes –> forAll ( a : Attribute | a@pre.visibility = #private
           implies a.visibility = #public) and
6      attributes –> forAll ( a : Attribute |
7          a.owner.operations –> exists (op : Operation |
               op.name = "get" + a.name) and
8          a.owner.operations –> exists(op : Operation |
               op.name = "set" + a.name)))
```

Figure 7: OCL specification of the `addAccessors` operation

Figure 7 depicts the OCL specification of the `addAccessors` operation. The precondition (line 2) is empty, which means that this transformation can be applied on any UML class diagram. The postcondition defines constraints on the UML class diagram that have to be respected after the transformation execution. It can be seen as the contract of the `addAccessors` transformation. It specifies that the following conditions have to be guaranteed for every classes of a class diagram:

- classes only contain attributes which visibility is private (line 4),
- if a class attribute visibility was public, it becomes private (line 5), and
- for any class attribute, there exists in the class, a method which name begins with "get" and ends with the attribute name, *i.e.* a getter (line 7), and a method which name begins with "set" and ends with the attribute name, *i.e* a setter (line 8)[4].

Constraints of lines 4, 7 and 8 are already defined by the target metamodel. Then, there are not mandatory in the postcondition of the `addAccessor` operation. However, line 5 express constraints on the evolution of the model elements during the transformation. It states that attributes that was private must become public. It it important to specify this because a class diagram on which all private attributes have been removed and containing only public

[4]In order to simplify the specification, the parameters and return type of the setter and the getter methods are not checked. In order for the specification to be complete, this should be done.

attributes with for each a getter and a setter method is conforming to the target metamodel. However, this is not a valid way to transform the model. For expressing that all attributes must be conserved, we must define a constraint on the evolution of model elements such as with the one of line 5.

We also need to express that the transformation operation constraints must be verified for each class of the target model. This can be done in the following way in OCL, even if this solution is not very elegant:

context Class::globalAddAccessors()
post: Class.allInstances() –> forAll (c | c^addAccessors())

3.3 The Proxy Addition Transformation Example

In the section, we study the contract definition of a more complex transformation as the previous section one. It notably emphasizes the problem of finding the right metamodel for the transformation operation specification.

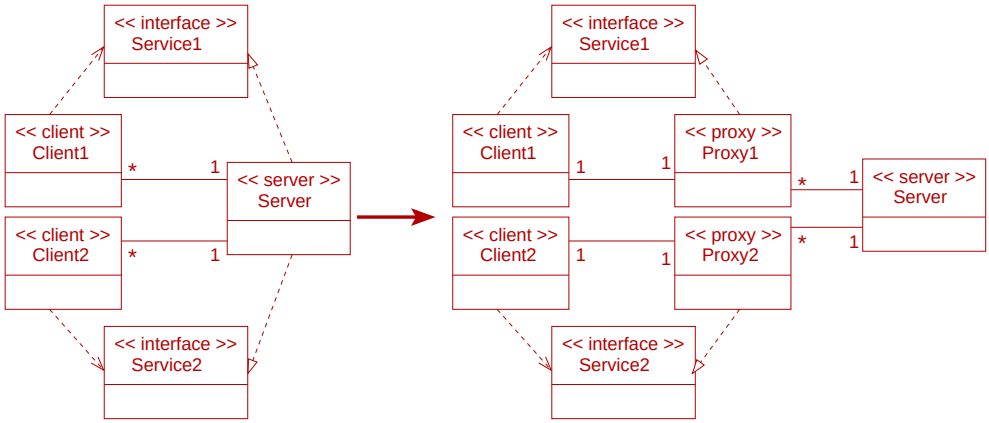

Figure 8: Addition of proxies between a server and clients

Figure 8 depicts a transformation aimed at introducing a proxy between clients and a server defined in a model. In this example, two kinds of clients are considered: the *Client1* class uses services defined in the *Service1* interface, and *Client2* uses services from *Service2*. The transformation introduces a specific proxy with each kind of client. Each proxy implements the service interface used by the client it is associated with.

The contract associated with this `proxyAddition` transformation defines three sets of constraints as we propose:

- Constraints on the source model: we impose the presence of clients, servers and interfaces with specific relationships among them for the model to be valid for transformation.
- Constraints on the target model: we impose the presence of clients, servers, proxies and interfaces with specific relationships among them for the model to be considered as a valid result of the transformation execution.
- Relationships between elements of the source and the target model: we impose that relationships among clients and servers through interfaces remain unchanged after the addition of proxies.

3.3.1 Definition of Profiles for Target and Source Models

Servers, clients and proxies are defined as stereotyped classes, using profiles, as shown on figure 9. The ClientServer profile defines the stereotypes for source models and the ClientProxyServer defines the stereotypes for target models. These profiles define that the three stereotypes are specialization of the Class class of the UML meta-model. The ClientProxyServer profile is an extension of the ClientServer profile: in addition to the concepts of client and server that are the same in both profiles, it defines the proxy concept.

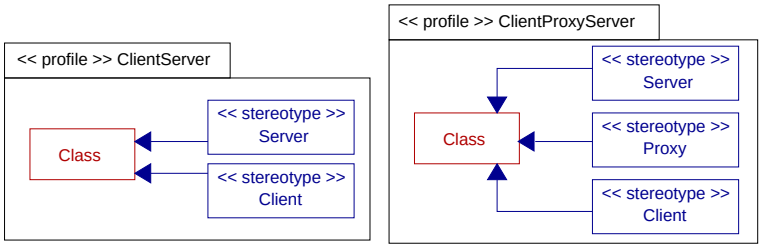

Figure 9: Specializations of the UML metamodel using profiles

In order to complete the definition of each profile, we have to define additional constraints on the profiled UML metamodels. To do so, we can specify invariants written in OCL for each profile.

```
context Class def: hasClassRefWith(cl : Class) : Boolean=
self.associationEnd.association.associationEnd.class
    -> exists ( c | c = cl )

context Client::newProxy() : Proxy
post: result.oclIsNew()
```

Figure 10: Definition of utility operations

```
1   context Client inv:
2   let servers = self.associationEnd.association.associationEnd.class
        -> select ( c | c.isTypeOf(Server)) in
3   let interfaces = self.dependsOn in
4   servers -> notEmpty() and
5   interfaces -> notEmpty() and
6   interfaces -> forAll ( i | servers.implements -> includes (i) and
7       servers -> forAll ( s | s.implements -> includes (i)
            implies s.hasClassRefWith(self)))
```

Figure 11: OCL invariants of the ClientServer profile

Figure 11 defines invariants for the ClientServer profile. It states that each client must be associated with at least one server (line 4) and must depend on at least one interface (line 5). For each interface a client is depending on (line 6), at least one server implementing this

interface must be associated with the client (line 6). Moreover, each server implementing this interface must be associated with the client (line 7). For simplifying the definition of this constraints, we use an operation helper for model navigation called `hasClassRefWith` and defined on figure 10.

All these OCL constraints define association rules among clients and servers through interfaces of services offered by servers. In the same way, we have to define association rules among clients, servers and proxies for the ClientProxyServer profile, *i.e.* the target metamodel.

```
1   context Client inv:
2   let servers = self.associationEnd.association.associationEnd.class
        -> select ( c | c.isTypeOf(Server)) in
3   let proxies = self.associationEnd.association.associationEnd.class
        -> select ( c | c.isTypeOf(Proxy)) in
4   let interfaces = self.dependsOn in
5   servers -> isEmpty() and
6   proxies -> notEmpty() and
7   interfaces -> notEmpty() and
8   interfaces -> forAll ( i | proxies.implements -> includes (i) and
9       proxies -> forAll ( p | p.implements -> includes (i)
            implies p.hasClassRefWith(self)))

10  context Proxy inv:
11  self.association.associationEnd.association.class -> select ( c |
        c.isKindOf(Server)) -> size() = 1
```

Figure 12: OCL invariants of the ClientProxyServer profile

Figure 12 defines these rules with OCL constraints. They state that each client must not be associated with any server (line 5), but must be associated with at least one proxy (line 6) and must depend at least on one interface (line 7). Each interface a client is depending on (line 8) must be implemented by a proxy associated with the client and if a proxy implements this interface, it must be associated with the client. Finally, each proxy must be associated with one and only one server (line 11).

3.3.2 Definition of the `proxyAddition()` Transformation Operation

Before defining in OCL the transformation operation, we have to answer two questions:

1. What is the most specialized metamodel for which both source and target models conform to?

2. What is the classifier of this metamodel owning the transformation operation?

We are here in the context of figure 3, presented in the previous section. The metalevel specializations are the profiles augmented with their respective OCL constraints. Each profile is a specialization of the UML metamodel. The *M2s* source metamodel is the ClientServer profile (including OCL invariants of figure 11) and the *M2s'* target metamodel is the ClientProxyServer profile (including OCL invariants of figure 12).

As explained in previous section, there are contradictions between the *M2s* complete source metamodel and the *M2s'* complete target metamodel (the first one express that a Client must

be associated with a Server and the second one that a Client must not be associated with a Server). Thus, we can not use any of these metamodels as context for the OCL operation transformation definition.

We propose to define simplified source and target metamodels as profiles but without their associated OCL constraints. The *M2'* target metamodel is then simply the ClientProxyServer profile, without the OCL constraints of figure 12. This metamodel is specialized enough for containing the operation transformation specification because it defines all specific concepts of both source and target models (client, server and proxy). So, these models can all conform to this metamodel. Other choices, notably in adding some OCL constraints on metamodels could have been taken, but this one as the advantage of simplicity.

We must now answer the second question. We propose to attach the transformation operation to the Client class because it will simplify the writing of OCL expressions.

```
1   context Client::proxyAddition()
2   post: self.dependsOn -> forAll ( i |
3           let proxy = self.newProxy() in
4           let server = i@pre.isImplementedBy in
            -- the "client@pre" is equals to self, no variable is required
5           server.isTypeOf(Server) and
6           proxy.implements -> includes (i) and
7           proxy.hasClassRefWith(self) and
8           proxy.hasClassRefWith(server) )
```

Figure 13: Specification in OCL of the `proxyAddition()` transformation operation

Figure 13 specifies in OCL this `proxyAddition()` operation attached to the Client class of the ClientProxyServer profile. In this specification we have to define that a proxy is associated between each client and a server and that this proxy implements now the interface of services the client is depending on. Constraints of figure 12 impose major constraints on relationships among these classes but it does not ensure that a client is still depending on the same interface[5].

So, for each interface a client is depending on (line 2), a new proxy is created (we use an utility operation defined on figure 10 to be able to reference this new proxy in a variable, even if this solution is not elegant[6]) and must be associated with the client (line 7), with the server the client was associated with (line 8) and must implement the interface (line 6).

We also need to express that the transformation operation constraints must be verified for each Client class of the target model. This can be done in the following way in OCL, even if this solution is not very elegant:

```
context Client::globalProxyAddition()
post: Client.allInstances() -> forAll ( c | c^proxyAddition())
```

[5]If we take the example of figure 8, a target model where interfaces have been swapped, and then where *Client1* is associated with *Interface2* and *Proxy1*, respects constraints defined in the target metamodel. However we can not consider this model as a valid result of the transformation. To define that the *Client1* must still depend on *Interface1*, we must constraint the way the proxy is added for each client. This can only be done in the transformation operation specification.

[6]Another solution would have been to define that the `proxyAddition()` operation returns a proxy, it would then be handled under the `result` OCL feature. But we do not really need or want to define an operation returning a proxy in our context.

4 Proposition of OCL Extensions

Through the discussion about the use of OCL for transformation operation specification and the example of proxy addition, one can notice that OCL suffers from some drawbacks and is not necessary well designed for some parts of transformation contract definition.

OCL is well-suited for defining precisely constraints and invariants at the metamodel level. However, navigation in the metamodel can become rapidly complex in some cases. Even in the context of our very simplified metamodel, some navigation expressions are a bit long to write. A solution to limit this problem would be to offer a set of utility operations for simplifying the navigation at the metamodel level (in the same way as we define the `hasClassRefWith` operation).

For the definition of the transformation operation with pre and post-conditions, there are two main problems. The first one is to find a metamodel for which both source and target models can conform to. The second one is the requirement to attach the transformation operation to a given classifier of a class diagram even if we want to specify an operation transforming a whole diagram.

To solve the first problem, a solution is to introduce a new feature allowing to reference both source and target models without the requirements for them to be instances of the same metamodel. For example, defining a new statement such as "**target**:" for referencing the model after the transformation and a "`@source`" construction allowing to reference element of the source model at the same time. We also need a construction for defining that an element of the source model matches an element of the target model (such as the $<\sim>$ operator defined in [8]). This would then enable transformation contract for very different source and target metamodels to be defined. Indeed, a major drawback of specifying the transformation operation using standard OCL is that source and target metamodels must have strong similarities.

Concerning the second problem, it would be interesting of being able to attach OCL expressions to packages or diagrams instead of only classifiers. This is useful for defining transformation operations applying directly on a whole diagram (without the requirement to express that an operation defined in a classifier must be applied for all instances of this classifier).

5 Related Works

In the past few years, an important amount of effort have been devoted to the study of transformation techniques. As result, numerous proposal have emerged. [6] gives a detailed review of existing approaches and their classification. All these techniques relies to the execution of transformations. We have not found in the literature techniques for specifying a transformation under the form of a contract as we propose in this paper.

The interest for model transformation engines is partially driven by the OMG Request for Proposal on Query/Views/Transformations (QVT) [11]. It promises to deliver a standardized way for transforming models. However, the proposal is not finalized yet and several responses have been submitted. Major players such as IBM, Alcatel-Thales-Softeam, or SUN submitted an answer and the final characteristics of the QVT are still unsure. However, several tools, not necessarily related to QVT, are already available for testing.

Like for programming languages these transformation tools or techniques can be separated in two categories: imperative and declarative approaches. As both have pros and cons, a third

category, called hybrid, mixes the two. TRL [1], Altas [5], or XDE [16] are examples of hybrid approaches.

Imperative approaches allow one to express transformations with a syntax rather similar to the one of a programming language (Java, Python, C++, etc.). Such approaches generally navigate a source model and create a target model according to elements found in the source model. The transformation code can be, either any piece of code, or, as in the Jamda [4] framework, can be structured with the visitor design pattern.

Declarative approaches are based upon the use of patterns in source and target models and their relationships. Whenever a source pattern is met, its associated target pattern is generated. For instance, the TRL approach allows to define rules to map source elements constrained by an invariant with target elements. XDE uses parameterized collaboration diagrams as patterns of models. Atlas is another example of source pattern to target pattern transformation.

The way to define a transformation contract can be sometimes rather similar to write the transformation in a declarative language. However, even if the form will be close, the goal is different: a contract defines the *what* of a transformation and is not executable by a tool.

Regarding a straight comparison with our approach, some works such as [15] have already studied the use of OCL in the specification of model transformations. However, they are not based upon plain OCL use and, as other approaches, their goal is to define executable transformations and not transformation contracts. For instance, [15] proposes to extend OCL for an operation specification with an action statement in addition to pre and postconditions. Its goal is to add imperative expressions in addition to the declarative part formed by the pre and postconditions. Several propositions to the QVT RFP also use OCL, generally as a query language, to find patterns of elements to be transformed. We use OCL in our contracts as a constraint language.

Kleppe and Warmer have been strongly involved in definition and evolution of OCL [17]. In [8] they propose a model transformation language model partly based on OCL. Here too, they propose a language for defining executable transformations.

Managing metamodels is somehow inherent to model transformations. Then, some works, such as [7], focus on the specification of transformations at the metamodel level. [3] proposes to define a metamodel of transformations. Here also, the goal of these approaches is to be able to define executable transformations.

In summary, most of the works around model transformations deal with defining languages and tools for executing transformations. Few works seems to be dedicated to the definition of transformation contracts.

6 Conclusion

Model transformation contracts aim at defining what a model transformation does, under which conditions it can be applied and what its expected result is. These contracts are important in the model-driven engineering processes, they would help in specification and documentation of transformations, validation and test of models and transformations and also in checking that a set of transformations can be executed in series on a model.

We have defined a transformation contract as a tuple of three elements:

- a set of constraints to be matched for a model to be candidate as a source model of the transformation,

- a set of constraints to be matched for a model to be considered as a valid target model produced by the transformation, and
- a set of constraints on the evolution of elements from the source to the target model

They are several ways and techniques to define these three sets. In the paper, we focus on standard UML and OCL features to define them. We also focus on transformations of UML class diagrams into UML class diagrams but of course transformation contracts can be defined for any kind of model or diagram. Constraints on source and target models are defined at the UML metamodel level by defining profiles augmented with OCL invariants. Relationships between elements of source and target models can be expressed in OCL with the specification of the transformation operation. However, this solution suffers from some drawbacks like imposing for source and target metamodels to have "enough similarities" and dependencies. It is then difficult to define a contract for models with completely or very different metamodels. We have given some hints and ideas to extend OCL in order to bypass this problem.

In the future, we plan to study others examples of transformations, including transformations of any kind of UML diagrams such as sequence diagrams or statecharts. We will also study transformation contract definition between two different technological domains or spaces (for instance, the transformation of a UML model into Java code). This will implies to clearly define and specify the OCL extensions we discussed in section 4.

We also plan to build or use tools (such as UMLAUT NG[7]) to validate models againts transformation contracts. This includes checking if a model can be transformed by a given transformation (i.e. if it respects constraints on source model) and if a model can be considered as a valid result of a transformation (i.e. if it respects constraints on target model and on relationships with the source model).

References

[1] Alcatel, Softeam, Thales, TNI-Valiosys, Codagen Corporation, LIFL, et al. *MOF Query/Views/Transformations Submission*, 2003. OMG Document ad/2003-08-05.

[2] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.

[3] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective Model Driven Engineering. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications*, volume 2863 of *LNCS*. Springer, 2003.

[4] P. Boocock. The Jamda Project, 2003. `http://jamda.sourceforge.net`.

[5] J. Bézivin, G. Dupé, F. Jouault, and J. Rougui. First Experiments with the ATL Model Transformation Language. In *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of MDA*, 2003.

[6] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of MDA*, 2003.

[7] `http://www.irisa.fr/UMLAUT/`

[7] S. R. Judson, R. B. France, and D. L. Carver. Specifying Model Transformations at the Metamodel Level. In *Workshop in Software Model Engineering (WISME) - UML 2003*, 2003.

[8] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained, the Model Driven Architecture: Practise and Promise*. Addison-Wesley, 2003.

[9] R. Marvie. Composing Transformation to Drive Modeling. Technical report, LIFL, 2004.

[10] B. Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, October 1992.

[11] Object Management Group. *Query/Views/Transformations RFP*, Apr. 2002. OMG Document ad/2002-04-10.

[12] OMG. *CORBA/IIOP 3.0.1 Specification*. Object Management Group, December 2002.

[13] OMG. UML 2.0 OCL 2nd revised submission, version 1.6, January 6, 2003, 2003. `http://www.omg.org/cgi-bin/doc?ad/2003-01-07`.

[14] OMG. UML 2.0 Specifications. `http://www.omg.org/uml/`, 2003.

[15] D. Pollet, D. Vojtisek, and J.-M. Jézéquel. OCL as a Core UML Transformation Language. In *WITUML: Workshop on Integration and Transformation of UML models, ECOOP 2002*, 2002.

[16] Rational. Xde. `http://www.rational.com/products/xde`.

[17] J. Warmer and A. Kleppe. *The Object Constraint Language – Second Edition, Getting Your Models Ready for MDA*. Addison-Wesley, 2003.