OCL for the Specification of Model Transformation Contracts

Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien

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

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 the specification of model transformation contracts. 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 the relevance and limits of standard OCL to define transformation contracts.

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 Platform Specific Models (PSM) level models, according to technical choices.

One major challenge of the MDA is 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). It promises to deliver a standardized support for model transformation [11].

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* [2,7] 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*. 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 by specifying and documenting transformations, validating and testing models and transformations, and also validating that a set of transformations can be executed in series on a model.

There are several ways for defining these transformation contracts. In this paper we investigate and discuss the relevance of OCL in this context. Indeed, we choose to define model transformation contracts using standard modeling or specification languages only, instead of creating a new language for this purpose. Even if OCL has now become a multi-purpose language, its basis is still to define constraints on models. It is then well suited for defining contracts that are composed of a set of constraints that must be respected by modeling or software elements.

The rest of this paper is organized as follows. Section 2 gives a general definition of model transformation contracts. Section 3 investigates the way to express them by using regular UML and OCL features. Section 4 gives a complete example of a model transformation contract. Section 5 proposes OCL extensions to improve and facilitate contract transformation specifications. Section 6 discusses related works, before concluding with some future trends.

2 Model Transformation Contracts

The goal of a contract is to define the *what* and not the *how* [2,7] 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, model transformations are applied to a source model and produce a target model. The transformation itself is defined as an operation. A *transformation contract* specifies those three 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 to a model and what its excepted result is. Such information are also useful for choosing and applying the proper transformation in a context of off-the-shelf transformations.

Validation and test Using adequate tools, it becomes 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 a basis for testing models and transformations—they can be used for defining oracles.

Chaining of transformations The MDA implies to execute multiple transformations to go from a PIM to the code. 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.

2.1 Study of a Transformation

Before giving a precise definition of transformation contracts, we show information required to be expressed in a contract through the study of an example.

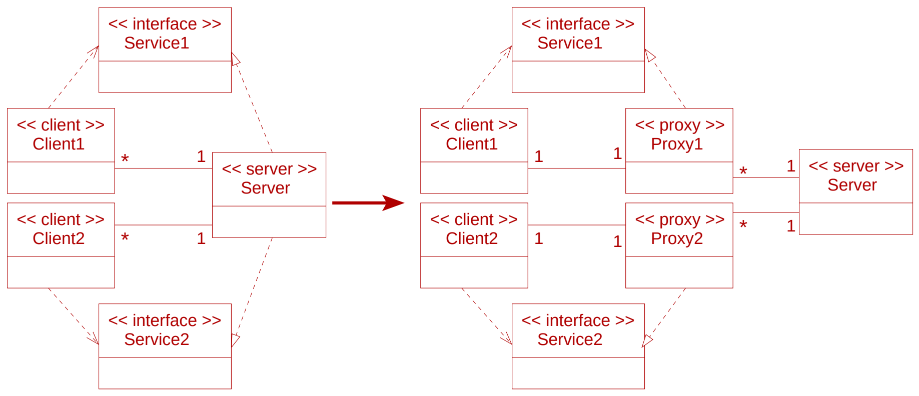

Fig. 1. Insertion of proxies between a server and clients

Figure 1 depicts a transformation aimed at introducing a proxy between clients and a server defined in a model of a software architecture. 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* one. The transformation introduces a particular proxy for each kind of client. Each proxy implements the service interface used by the client it is associated with[1].

The contract of this transformation deals with expressing what the transformation does and under which conditions it can be applied. First of all, this transformation can only be applied on class diagrams defining clients, interfaces and servers with appropriate relationships among these elements. The contract must specify these constraints on source models. Moreover, similar constraints have to be defined for a target model to be considered as a valid result of the

[1] We use stereotyped classes to differentiate clients, servers and proxies. This is only for simplifying the example. Our methodology of contract specification is not based on stereotyped classes, it could be applied to any kind of UML diagrams without particular constraints.

transformation execution: we impose the presence of clients, servers, proxies and interfaces with specific relationships among them.

Defining constraints that must be matched by source and target models is not enough for completely specifying this transformation. In our example, the contract must define that a client on the source model is associated with a server through an interface. In the meantime, a client on the target model has to be associated with a proxy through an interface. Given independently, these two constraints do not prevent a couple of source and target models to be invalid regarding the aim of the transformation—even if they respect constraints on source and target models.

For instance, if we consider the target model of our example (right diagram on figure 1) for which *Interface1* and *Interface2* have been swapped, this diagram respect constraints on target models. A client is associated with a proxy through an interface. But for this target model to be valid, we must specify that the interface a client is depending on and that its associated proxy is implementing, is the one this client was depending on in the source model. The transformation contract must then be able to express relationships between elements of both source and target models.

2.2 Definition of Model Transformation Contracts

Given the above example, we know what a model transformation contract must contain two kinds of constraints: First, it must specify constraints on source and target models and second, it must define constraints on relationships between source and target model elements.

As a definition, a transformation contract is 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 relationships and evolution of elements from the source to the target model.

3 Specification of Model Transformation Contracts using UML and OCL

There are several ways and techniques to define model transformation contracts. We have chosen to define them using as much as possible standard modeling or specification languages. Our motivation was to avoid the definition of a new language for this purpose. We first investigate the use of standard UML [9] and OCL [13] for defining transformation contracts. Our first experimentations have been restricted to the transformation of UML class diagrams. However, this approach can be applied to other UML diagrams such as sequence or state ones and even to MOF models.

OCL in version 2.0 [8] is now becoming a multi-purpose language. It can be used for defining constraints of course, but it is also a powerful querying language. However, its basis is still to define constraints on models. It is then well suited for defining contracts that are composed of a set of constraints that must be respected by modeling or software elements. In this section, we show that specifying precise constraints on source and target models is straightforward in OCL at the metamodel level. However, specifying constraints on relationships between source and target model elements is not as simple.

3.1 Definition of Constraints on Source and Target 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. For instance, adding constraints on a class diagram is done by defining constraints on the metamodel, specifying the structure and elements of class diagrams. So, constraints on source and target models can be expressed by constraining through a specialization of the UML metamodel. This can be done using a profile as proposed in the UML specification [9]. Such a profile defines concepts specific to a given modeling context as well as relationships among these concepts and standard UML features. In order to precisely and fully define constraints attached to profile elements, invariants written in OCL have to be specified.

For instance, a profile can define the concepts of the previous example: *Client*, *Server* and *Proxy*. These stereotyped classes are defined as extensions of the Class class defined in the UML metamodel. Specifying precise relationship constraints among these concepts is achieved by writing OCL expressions on the specialized metamodel within this profile.

3.2 Definition of Constraints on Relationships between Source and Target Model Elements

In order to express constraints dealing with relationships between source and target model elements, we must be able to reference both source and target models in OCL expressions. We have identified two ways for defining such OCL expressions:

– specifying in OCL the transformation operation through pre- and post-conditions in a "standard way",
– Writing OCL expressions that can directly handle both source and target models under the form of packages.

Each method presents both advantages and drawbacks. We discuss the first method in [3] and present the second one in this paper.

Specifying the Transformation Operation with Pre and Post-Conditions. The idea is to define a transformation operation in such a way that the precondition statement deals with the state of the model before the transformation, *i.e.* the source model, and the post-condition with the state of the model after the transformation, *i.e.* the target model. The OCL `@pre` construction enables model elements before the transformation operation call to be referenced in the postcondition specification. Source model elements are then handleable in the postcondition. So, in the postcondition, both source and target models are referenced which enables relationships between elements of both models to be defined.

For the proxy addition example we describe above, here is the main form of the OCL transformation operation specification:

```
context Client::proxyAddition()
post: -- constraints on the target model and source model elements (with the
      -- @pre OCL construction)
```

This solution is elegant in the sense that it corresponds to the "standard way" of specifying an operation using OCL. It is then somehow logical to try to express a transformation operation using this standard approach.

However, this solution has major drawbacks. It notably implies that both source and target models conform to the same (or close enough) metamodel. This major constraint leads to the fact that only particular transformations can be specified—such as model refinement, where only elements are added during the transformation. Transformations involving source and target models conforming each one to very different metamodels cannot be specified using this approach. For instance, it is not possible to specify a transformation of a UML class diagram into a Java program because the UML and Java metamodels are completely different.

Transformation operations defined this way must be attached to a classifier of the common metamodel. Often, this allows a simplification of the writing of some OCL expressions with shorter navigation on the metamodel by choosing the more adequate classifier.

Another advantage of this first solution is to keep an implicit relation between the source and the target model. Indeed, we can decide for instance that all source model elements and their relationships that are not constrained by the post-condition are kept unchanged in the target model as they was[2]. This avoid the writing of some extra OCL expressions.

Allowing Direct Handling of Both Source and Target Model Elements. A simple way to realize a direct handling of both source and target model elements is to define constraints against elements of two packages. One package contains the source model and the other one the target model. The OCL expressions are such that each package is directly handleable in these expressions.

[2] Under the condition that it remains coherent with the constraints on the target models as defining in the target profile.

Then, there are no particular constraints on the choice of the classifier on which these OCL expressions are attached. A simple way to apply this second approach is to define the transformation operation in the following manner:

```
context Package::proxyAddition(Package source) : Package
post: -- source parameter: references the source model
      -- OCL result pseudo-variable: references the target model
```

The transformation operation is attached to the Package metaclass or any other classifier. It takes the source model as parameter and returns a model corresponding to the target model. OCL expressions defining constraints and relationships among source and target model elements are then defined in the post-condition.

In the next section, we describe a full example of a transformation contract following this second approach underlying advantages and drawbacks of this method. First of all, compared to the first solution described above, the main advantage is to be able to define the contract of any kind of transformation, including the ones with very different source and target metamodels.

The main drawback is that we need to be able to express that an element of a source model is the same element or is mapped onto an element of the target model. For instance, a UML class can be mapped onto a Java class when specifying a transformation of a UML class diagram onto a Java program. OCL does not permit to express these mappings and must then be extended with a new construction (such as the $<\sim>$ operator defined in [5]).

The other drawback is that we need also to express all necessary mappings or equalities among source and target elements and their relationships. In the context of the first solution, this mapping is more or less implicit. Indeed, by default, it is possible to consider that all relationships and elements that are not constrained by OCL expressions are kept as they were.

Comparison of the Two Methods. Here is a synopsis of both method advantages and drawbacks:

1. Specifying the transformation operation in a standard OCL way
   - Pros:
     - Implicit mapping among source and target model elements
     - Simplification of the OCL expressions through the right choice of the classifier owning the operation transformation
   - Cons:
     - Do not permit all kinds of transformation to be specified
2. Direct handling of both source and target models
   - Pros:
     - Can define any kind of transformation
   - Cons:
     - Need an explicit mapping among source and target model elements and their relationships

- Need an OCL extension for defining this mapping
- More verbose

To summarize, the first method gives a simplified definition of relationships among source and target elements but cannot be used for all transformations. On the other hand, the second method can define the contract of any kind of transformation but is more verbose and implies more complex specifications.

4 Example of Contract Definition

This section presents the complete contract of the proxy addition transformation described in section 2.1. For specifying relationships between source and target model elements, the second approach we have presented above (direct handling of both source and target models) is used. The specification of the same contract using the first approach (standard specification using OCL of the transformation operation) is done in [3]. To begin with, we present our simplified UML metamodel. Its goal is only to simplify the writing of OCL expressions at the metamodel level.

4.1 A Simplified UML Metamodel

The UML 2.0 metamodel [9] 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 the transformation discussed in this paper—has been defined. In addition to removing all the irrelevant elements of the UML metamodel for our example, some elements and relationships have been slightly modified.

The simplified UML metamodel is defined by the class diagram depicted by figure 2. In order to be complete, the semantics of the metamodel elements should be given, by notably specifying in OCL the well-formedness rules.

4.2 Definition of Profiles for Target and Source Models

A transformation contract is composed of three sets of constraints. In this section, we specify the first two sets: on source and target models, *i.e*, profiles defining source and target metamodels[3].

Servers, clients and proxies are defined as stereotyped classes, using profiles, as shown on figure 3. The ClientServer profile defines the stereotypes for source models and the ClientProxyServer defines the stereotypes for target models. These profiles state that the three stereotypes are specialization of the Class class of

[3] These profiles are exactly the same as the ones in [3]. Indeed, the difference between the two approaches in the way to specify the contract is only related to the specification of the relationships between source and target model elements.

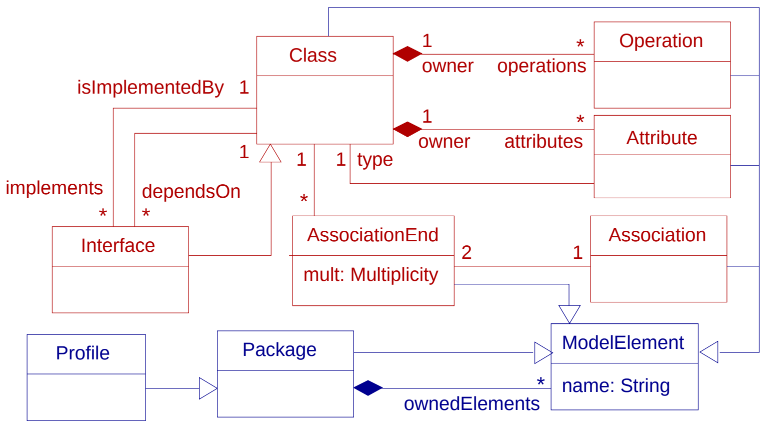

Fig. 2. Our simplified UML metamodel

the UML metamodel. The ClientProxyServer profile is a kind of 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.

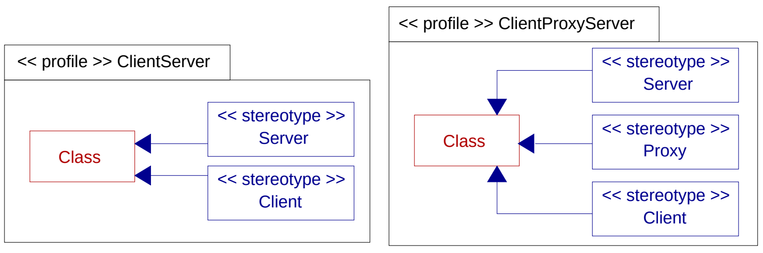

Fig. 3. Specializations of the UML metamodel using profiles

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

Figure 4 defines invariants for the ClientServer profile. It states that each client must be associated with at least one server (line 5) and must depend on at least one interface (line 6). For each interface a client is depending on, at least one server implementing this interface must be associated with the client (line 7). Moreover, each server implementing this interface must be associated with the client (line 8). For simplifying the definition of this constraint, we use an operation helper for model navigation called `hasClassRefWith` that returns true if the element on which it applies has an association with the element passed

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

Fig. 4. OCL invariants of the ClientServer profile

as parameter[4]. All these OCL constraints define association rules among clients and servers through interfaces of services offered by servers.

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

11  context Proxy inv:
12  self.association.associationEnd.association.class −> select ( c |
        c.oclIsKindOf(Server)) −> size() = 1
13  endpackage
```

Fig. 5. OCL invariants of the ClientProxyServer profile

In the same way, we have to define association rules among clients, servers and proxies for the ClientProxyServer profile, *i.e.* the target metamodel. Figure 5 defines these rules with OCL constraints. They state that each client must not be associated with any server (line 6), but must be associated with at least one proxy (line 7) and must depend on at least one interface (line 8). Each interface

[4] This operation is specified in OCL in [3], on figure 10, page 12.

a client is depending on (line 9) must be implemented by a proxy associated with the client. The proxy implementing this interface must also be associated with this client (line 10). Finally, each proxy must be associated with one and only one server (line 12).

4.3 Specification of the Transformation Operation

The last set of constraints of the contract deals with expressing relationships between source and target models. With the second approach we use, we need to express that an element of the source model has to be mapped onto an element of the target model. For this purpose, we propose to add the following extension to OCL: `ens1 -> mappedOnto (ens2)`
It expresses that each element of the `ens1` collection must map to an element of the `ens2` collection. Both collections have the same size and the mapping is from one element to one element.

A mapping can also be expressed for two model elements:
`elt1.mappedOnto(elt2)`
It must return true if there is a mapping between `elt1` and `elt2` elements, false otherwise. This second OCL extension can be seen as a boolean query operation of the ModelElement metaclass. Indeed, by construction of UML profiles, all profile or package elements are direct or indirect specializations of this metaclass. Then this operation can be applied to any kind of model element couples. For precisely defining the mapping constraints between elements in a given context, this operation can be specified using OCL.

Figure 6 specifies in OCL the `proxyAddition` transformation operation. It takes as parameter the source model (as a package) and returns the target model (also as a package). First of all, line 2 expresses that the source package must conform to the ClientServer package (or profile) and line 3 that the target model must conform to the ClientProxyServer one.

Line 4 expresses that each client of the source model has to map to a client of the target model[5]. Same kind of constraints could be added for servers.

Line 5 expresses that each element of a subset of the source model elements must either exist in the target model (as a clone) or be mapped onto an element of the target model. This expresses that elements of the source model are conserved as is or are mapped in the target model. Associations that were connecting servers to clients have not to respect these constraints because they are removed through the transformation[6] (this is the reason of the `reject` expression in

[5] This expression is here to show a simple usage of our mapping expression. Indeed, on both metamodels, Client classes are identical, by construction of the profiles. Then we could instead specify that clients of the target model are clones of the ones of the source model. We can use for that the equality symbol "=" to express that two model elements are identical in both packages (this also includes that their metaclasses are identical). For less ambiguity, we could add a `isCloneOf` OCL operation to express these constraints of equality.

[6] AssociationEnd elements associated with these associations should also be removed of the set. This is not specified here for simplifying the operation specification.

```
1  context Package::proxyAddition(Package source) : Package
2  pre: source.oclIsTypeOf(Profile) and source.name="ClientServer"
   post:
3  result.oclIsTypeOf(Profile) and result.name="ClientProxyServer" and
4  source.modelElement –> select ( me | me.oclIsKindOf(Client)) ->
       mappedOnto ( result.modelElement –> select ( me | me.isKinfOf(Client))) and
5  source.modelElement –> reject( me | me.oclIsKindOf(Association) and
       me.oclAsType(Association).connectClientServer())
           –> forAll ( me | result.modelElement –> exists( me2 |
               me2 = me or me2.mappedOnto(me))) and
6  result.modelElement –> select ( me | me.oclIsKindOf(Client)) –>
       forAll ( cl | cl.oclAsType(Client).dependsOn –> forAll ( i |
7          let sourceServer = (source.modelElement –> select ( me |
               me.oclIsKindOf(Interface) and me.oclAsType(Interface) = i))
                   –> first().oclAsType(Interface).isImplementedBy in
8          let server = (result.modelElement –> select ( me |
               me.mappedOnto(sourceServer)) –> first() in
9          let proxy = self.newProxy() in
10         server.oclIsTypeOf(Server) and
11         proxy.implements –> includes (i) and
12         proxy.hasClassRefWith(cl) and
13         proxy.hasClassRefWith(server) ) )
```

Fig. 6. Specification in OCL of the `proxyAddition()` transformation operation

which `connectClientServer()` is a primitive returning true if the association is connecting a client to a server; this primitive, not shown here due to the lack of place, is also specified in OCL).

The rest of the specification deals with defining that after the transformation, as we explain in section 2.1, clients are still depending on the same interface. For each interface a client is depending on (line 6), a new proxy is created (we use an utility operation[4] to be able to reference this new proxy in a variable, even if this solution is not elegant) and must be associated with the client (lines 9 and 12). The proxy must implement the interface (line 11). This proxy must also be associated with the server the client is associated with (line 13). This server is the model element that maps the server the client was depending on in the source model through the interface (lines 7 and 8).

5 OCL Extensions and OCL Expression Simplification

Through the discussion about the use of OCL for transformation operation specifications 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 precisely defining constraints and invariants at the metamodel level. However, navigation over the metamodel can become rapidly complex in some cases. Even in the context of our simplified metamodel, some navigation expressions are a bit verbose to write. A solution in order to limit this problem would be to propose for a metamodel a set of utility operations for simplifying the navigation at the metamodel level (in the same way as we define the `hasClassRefWith` operation for our UML metamodel). The QVT merge group revised submission [11] also proposes to simplify the syntax of some OCL expressions when they are too verbose.

A problem we have to face when defining a contract (using the second approach) is to express that an element of the target model is a mapping of a source model element. As proposed in section 4.3, a mapping operator has to be added to OCL for this purpose.

6 Related Works

In the past few years, an important amount of effort has been devoted to the study of transformation techniques. As result, numerous proposal and transformation engines have emerged. [4] gives a detailed review of existing approaches and their classification. The current interest for model transformation is partially driven by the OMG Request for Proposal on Query/Views/Transformations (QVT) [10] which promises to deliver a standardized way for transforming models. A merge of initial submissions to this RFP is on the way to be standardized [11]. All these techniques rely on the evaluation of transformations. We have not found in the literature techniques for specifying a model transformation using a contract as we propose in this paper.

However, the QVT merge group submission [11] proposes to define transformations through relations and mappings. A relation defines a transformation in a specification purpose and a mapping is a realization of a relation in an executable purpose. QVT relations seems to be a good basis in specifying model transformation contracts and could be complementary to our pure OCL approach.

6.1 OCL Use in Transformation Techniques

The QVT merge group submission proposes to use OCL as a query language and to define an OCL-based language for describing patterns of model elements to be transformed. Other works such as [12] have also 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.

Kleppe and Warmer have been strongly involved in the definition and evolution of OCL [13]. In [5] they propose a model transformation language model partly based on OCL. Here too, they propose a language for defining executable transformations.

Finally, OCL is often used as a query language in transformation techniques. However we have not found works using OCL as a constraint language for defining model transformation contracts as we do.

6.2 Alternatives to OCL as Contract Definition Language

In our context of modeling using standards such as UML, OCL is a natural base for specifying model transformation contracts. However, it is of course possible to use other techniques for defining these contracts. For instance, B [1] is a powerful and well-known method for building validated software through formal specifications and refinement. Some recent works deal with applying B to UML diagrams and specifications. It would be interesting to see how model transformation contracts could be defined in this context.

On a more implementation-oriented side, the Java Modeling Language (JML [6]) allows definition of contracts on Java programs or Java-based interfaces. Using JML for defining model transformation contracts is another interesting way to explore; notably if tools realized for validating models and transformations are written in Java.

7 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 model-driven engineering processes: They would improve the specification and documentation of transformations, the validation and test of models and transformations, and also in checking that a set of transformations can be executed in sequence 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 relationships between elements of source and target model elements.

There are several ways and techniques to define these three sets. In this paper, we focus on standard UML and OCL features to do so. 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 models or diagrams. Constraints on source and target models are defined at the UML metamodel level by defining profiles enriched with OCL invariants. OCL is well-suited for precisely defining constraints on models. However, it suffers from too verbose statements. This could be solved by adding operations for navigating over a metamodel and simplifying the syntax of some OCL expressions, as also proposed by the QVT merge-group submission.

Relationships between elements of source and target models are a bit more difficult to express in OCL. We have described two approaches for this purpose. The first one uses "standard operation specification" in OCL but cannot be used for expressing every transformation contract. The second approach solves this problem but requires an extension of OCL for expressing mappings between models with different metamodels. This extension requires some work to be formally specified.

To summarize, OCL through a relatively small extension and some syntax and metamodel navigation expression simplifications, is a good basis in the specification of model transformation contracts.

In the future, we plan to study other examples of transformations, including transformations of any kind of UML diagrams such as sequence diagrams or statecharts. We will also study transformation contract definitions between two different technological domains or spaces. This will imply to clearly define and specify the OCL extension we discussed in section 4.3.

We also plan to build or use tools to validate models against transformation contracts. This includes checking if a model can be transformed using 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. J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.
2. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.
3. E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. Model Tranformation Contracts and their Definition in UML and OCL. Technical Report 2004-08, LIFL, April 2004.
4. 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.
5. A. Kleppe, J. Warmer, and W. Bast. *MDA Explained, the Model Driven Architecture: Practise and Promise*. Addison-Wesley, 2003.
6. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06x, Iowa State University, 2003.
7. B. Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, October 1992.
8. OMG. UML 2.0 OCL 2nd revised submission, version 1.6, January 6, 2003, 2003.
9. OMG. UML 2.0 Specifications. 2003.
10. OMG. *Query/Views/Transformations RFP*, 2004. OMG Document ad/2002-04-10.
11. OMG QVT-Merge Group. *Revised Submission for MOF 2.0 Query/Views/Transformations RFP*, 2004. version 1.6 (2004/08/16).
12. 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.
13. J. Warmer and A. Kleppe. *The Object Constraint Language – Second Edition, Getting Your Models Ready for MDA*. Addison-Wesley, 2003.