OCL contracts for the verification of model transformations

Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam

Université de Pau et des Pays de l'Adour / LIUPPA
B.P. 1155, 64013 Pau Cedex France
{Eric.Cariou, Nicolas.Belloir, Franck.Barbier}@univ-pau.fr,
Nidal.Djemam@etud.univ-pau.fr

Abstract. A model-driven engineering process relies on a set of transformations which are usually sequentially executed, starting from an abstract level to produce code or a detailed implementation specification. These transformations may be entirely automated or may require manual intervention by designers. In this paper, we propose a method to verify that a transformation result is correct with respect to the transformation specification. This method both includes automated transformations and manual interventions. For that, we focus on transformation contracts written in OCL. This leads to making the proposed method independent of modeling and transformation tools. These contracts are partially or fully generated through a dedicated tool.

1 Introduction

A software development process based on model-driven engineering relies on a set of transformations. They are usually executed in series to start from an abstract level of modeling and to arrive at the final code or to a detailed implementation specification. These transformations can be entirely automated or require the manual intervention of designers. Indeed, even if the main goal of model-driven engineering is to automate a complete software process, designers must often intervene on models and make choices about particular actions to be carried out.

It is important to be able to guarantee that the realized transformations are valid, *i.e.* they respect the transformation specification. This is an even more important issue when designers deal manually with models. In this case, models can be extensively modified without particular constraints and it should be made sure that modifications occur within the required scope.

In [6,5], we established conceptual bases of specifying transformations through model transformation contracts written in OCL (Object Constraint Language [11]). A couple of models verify a transformation if they respects the associated contract. In this paper, we validate in practice this approach by providing a method and its implementation in the context of endogenous transformations (i.e., metamodels remain constant when transforming). Our method allows the contract verification to be carried out starting from two models – one representing the source model and the other the target model of the transformation – generated or obtained as output of any tool, including when designers process models manually. The choice of OCL as language expression contract is justified by the fact that OCL is usable within several technological spaces and is a standard relatively well known and accepted by model designers.

Compared to our previous work, we also deeply discuss key issues, such as the OCL contract expression context problem and mappings between elements of the source and the target models. We show that contract writing can be greatly facilitate thanks to automatically generated mapping functions through adequate tools. Finally, we point out the limitation of the single context for expressing OCL constraints, which leads to problems when manipulating simultanously several models like in the case of model transformations.

Section 2 recalls the concept of model transformation contract. Section 3 gives an example of a model transformation and of its contract. Section 4 presents the method of definition and verification of a contract and applies it to our example. It also presents dedicated tools that help in writing and verifying contracts in the context of the Eclipse/EMF platform[1]. Lastly, before the conclusion, we discuss related works and the choice of OCL as contract expression language.

2 Model transformation contracts

Programming and designing by contracts [7,2] consist in specifying what does a software component, a program or a model, in order to know how to properly use it. Design by contracts also allows the assessment of what has been computed with respect to the expressed contracts. In terms of computing operations, being operations at code level or operations in models, contracts are embodied in pre and post-conditions. A pre-condition defines the state of a system to be respected (if not, the system is in abnormal state) so that operations can be called. Post-conditions establish the state of a system to respect after calls. One can also specify invariants that have to be permanently respected. In contrast with pre and post-conditions, invariants are not expressed in relation with operations inputs and outputs.

In the research context about model transformations, we proposed in [6,5] to apply these principles to specific operations: those of model transformations. We thus want to express contracts on transformations themselves. These contracts describe expected model transformation behaviors. Formally, constraints on the state of a model before the transformation (source model) are described. Similar constraints on the state of the model after the transformation (target model) are required. Post-conditions guarantee that a target model is a valid result of a transformation with respect to a source model. Pre-conditions ensure that a source model can effectively be transformed.

A transformation contract is defined by three distinct sets of constraints:

Constraints on the source model : constraints to be respected by a model to be able to be transformed

[1] `http://www.eclipse.org/emf`

Constraints on the target model : general constraints (independently of the source model) to be respected by a model for being a valid result of the transformation

Constraints on element evolution : contraints to be respected on the evolution of elements between the source model and the target model, in order to ensure that the target model is the correct transformation result according to the source model contents

3 Model transformation example: interface addition

As an illustration, we study in this article a transformation that consists in a refinement wich adds interfaces to a UML class diagram. To simplify the writing of the associated contract, we provide a simplified UML meta-model because the original meta-model is relatively complex in terms of element number and of navigation possibilities between these elements. This simplified meta-model is exposed and discussed in the next section.

3.1 Meta-model of class diagrams

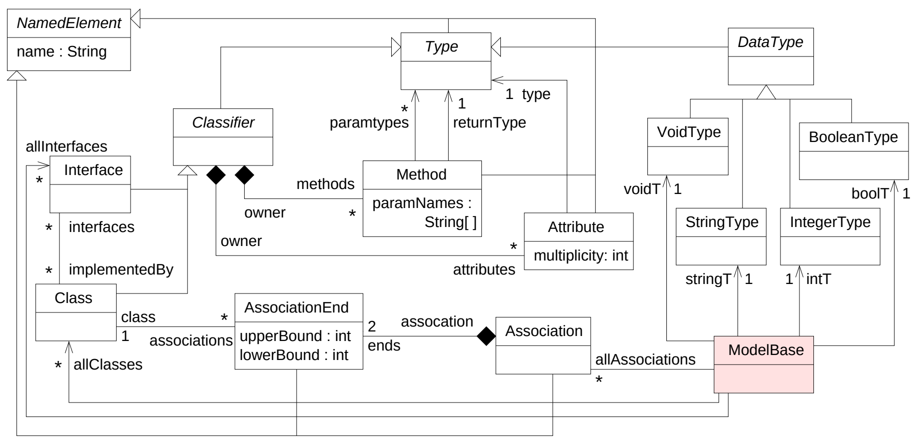

Fig. 1. Meta-model of simplified class diagrams

Figure 1 is the meta-model of simplified class diagrams. One finds there the same basic concepts as in a UML class diagram: class, interface, type, method, attribute, association as well as four primitive types (Void, Boolean, Integer and String). The concepts which are not present, by simplification, are for example visibilities or specializations. The `ModelBase` element (Fig. 1 on the right hand side, bottom) has a special role: it represents, as its name shows, the base of the model. It does not define a concept of the domain but is a help in navigating on the model by giving direct access to the whole set of the main model elements. This kind of element is not mandatory but it can be useful in model manipulation with some tools, particulary in the context of the Eclipse/EMF platform we used.

Some OCL constraints, the well-formedness rules, have to be added to supplement the class diagram of the meta-model, for instance to check that an interface does not have an attribute.

3.2 Example of refinement: addition of interfaces

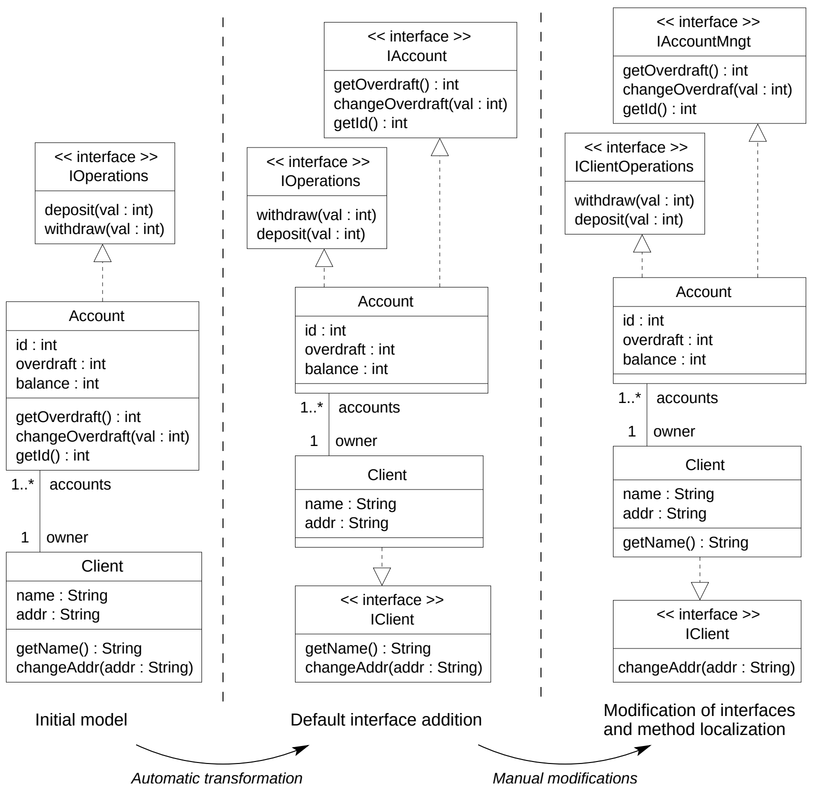

Fig. 2. Refinement example: addition and modification of interfaces

The principle of refinement used in our example is to add one or more interfaces to each class and to move a part or all the methods implemented by this class to its factorized interfaces. This refinement is carried out in two steps:

1. Automatic addition of an interface by default: for each class of the diagram, we create (except if it already existed) an interface named `I`*`className`*.

All the methods directly supported by the class are then moved to this interface. This operation is entirely automated and is realized by a model transformation tool.
2. Possible revised layout of methods and interfaces: the addition of an interface by default is not systematically appropriate. Designers may then manually modify for each class the list of its interfaces (renaming, suppression, addition) as well as the method localizations (moving of a method towards another interface or the initial class).

Finally, after these two steps, we obtain a target model which is the transformation of the source model and corresponds to an interface addition refinement. The main constraint to satisfy for having the refinement correct is that each class always implements the same list of methods, directly or through its interfaces. In a detailed way, the transformation contract associated with this refinement is the following:

– Constraints on the source model: none, any class diagram can be refined;
– Constraints on the target model: each class implements at least one interface (even if it does not contain methods);
– Constraints on element evolution from the source model towards the target model: all the classes are maintained and each implements, directly or indirectly through its interfaces, the same method list as before the transformation.

Figure 2 shows an example of such a refinement, with its two steps. We can notice that the designer has renamed the two interfaces of the `Account` class and that the `getName` method has been straightforwardly replaced in the `Client` class. We observe that the refinement is correct since all the classes are preserved and each class always implements, directly or indirectly, the same list of methods. Finally, each class implements at least one interface.

4 Transformation verification by contract

In this section, we present in details a method for contract definition and verification. This method is the most generic as possible. It must thus be applicable to two models, one supposed the source model and the other the target model of the transformation or the refinement, defined or obtained from a large variety of tools, including when the designer intervene manually on models.

4.1 Contract expression context for checking element evolution

In [6], we defined two techniques to write a transformation contract, and more precisely the constraints on the evolution of the elements between the source model and the target model. These constraints are special because they require at the same time to refer to the source model and to the target model. OCL constraints being expressed in a single context, it is then necessary to be able to manipulate both models from a single context.

Specification of the transformation operation To carry out this problem, the first technique consists in attaching the transformation operation to an element of the meta-model and to specify it in OCL: the model before the transformation is the source model and once the transformation has been executed, the model modified by the operation is the target model. Within the post-condition, we can refer to elements of the target model and of the source model via the OCL `@pre` operator. Thus, the pre-condition allows the contract's part relating to the source model to be written and the post-condition contains the two other parts of the contract (on the target model and on the element evolution). For our interface addition example, we could have a contract of the following form:

```
context ModelBase::addInterfaces()
pre: -- constraints on source model
post:
   -- constraints on target model and
   -- constraints on element evolution between source and target models
   allClasses -> size() = allClasses@pre -> size() and ...
```

Here, the transformation operation is called `addInterfaces` and is attached to the `ModelBase` meta-element. The post-condition of the operation checks particularly that the number of classes after the transformation is the same as before. We can see that one can manipulate together elements before the transformation (from the source model) and elements after the transformation (from the target model).

This method is conceptually relevant because it associates a contract with a transformation operation as one can associate a contract in OCL with any operation of a class (via pre and post-conditions). However it is not adapted to our problem, for two reasons. Firstly, this contract is checkable only when the transformation is executed. This implies that we need a tool which at the same time makes it possible to execute a transformation and to check the associated constraints. This strongly reduces the set of usable tools. The second reason is the consequence of the first: it is necessary to execute a transformation to validate its contract, which requires an entirely automatic execution. If the designer modifies by hand the generated model, it is thus impossible to validate the contract.

Concatenation of source and target models When defining the contract, to avoid the problems we described above, we use the second technique we defined. Its general principle is to concatenate the source model and the target model in a more global model. Next, we define in OCL a set of invariants associated with the global model, and thus at the same time with elements of the source model and the target model. This model concatenation is a "trick" required to overpass the OCL limitation of invariants being expressed for a single context and as a consequence, being checked on a single model.

In a previous step of this current work, we presented in [4] a concatenation method where the designer had to explicitely add on metamodels elements for realizing model concatenation. He had also to make by hand this model concatenation or to develop a dedicated program to do it for each kind of metamodel. This was not acceptable because, as far as possible, the model designer must not be compeled by this need of model concatenation for checking the transformation contracts.

In the context of the Eclipse/EMF platform, we have then developped a tool that automatically realizes the concatenation for any meta-model. The tool takes as input the meta-model, the source and the target model. In a first step, it adds to the meta-model a `ModelReference` class which contains a string attribute called `modelName`. All classes of the meta-model are modified to specialize this new class. In a second step, the tool adds all elements of the source model and all elements of the target model into a third global model conforming to the modified meta-model. During this step, each element is tagged through the `modelName` attribute with the "source" or "target" string value, depending on the model it belongs. As output, our tool returns the modified meta-model and the global model containing all elements of both source and target models with indication of their origin[2].

4.2 Definition of the contract

As shown, the contract is composed of three parts: the constraints the source model must respect, the constraints the target model must respect and the constraints on the evolution of the elements between the source model and the target model. The first two parts are easy to express, we have to define invariants associated with the meta-model, like this is done in a common way. For our example of interface addition, the source model is any kind of class diagram without particular restriction; no constraint on the source model has to be defined. For the target model, each class of the diagram has to implement at least one interface. This is expressed in the following way:

```
context Class inv hasAnInterface:
self.interfaces -> notEmpty()
```

A standard OCL evaluator can then be used to check that this constraint is validated by the target model.

The third part concerning the evolution constraints between the two models is more complex to establish, because it is associated with the global model which concatenates the source model and the target model within one single model. It

[2] We work here on transformations implying a single source model and a single target model. The concatenation principles we present are directly generalizable for managing more than one source or target model. One has just to give an unique name for each model to tag its elements. In a more general way, our complete method is directly generalizable to manage more than one target or source model.

is then necessary to determine explicitly references on the source model and on the target model and their elements. This is easy to do as all elements of the global model, including our `ModelBase` instances, are tagged in order to indicate if they belong to the source or the target model. Moreover, we need to express that an element on a model is mapping another element on the other model and to get this mapped element to be able to express evolution constraints between these elements. This can be achieved by a set of OCL utility functions.

Mapping between elements of source and target models In our refinement example, the main goal of the contract, concerning the evolution of the elements between the source and the target models, consists in checking that each class of the target model implements, directly or via its interfaces, the same methods as its equivalent class on the source model. To validate that, it is then necessary to be able to determine which is its equivalent class. In a more general way, it is necessary to be able to determine that a given element on a model has or has not an element of the same type which maps it on the other model, and to be able to get this mapping element if it exists. If we take the example of Figure 2, the class `Account` on the target model (the final model) maps the class `Account` on the source model (the initial model). This mapping is checked by comparing the names of the classes and by checking that their attributes (here `id`, `overdraft` and `balance`) are the same. It is thus necessary to also check the mapping of the attributes.

The mapping of an element of a model to an element of the same type on another model can be of three kinds:

Full mapping an element maps to an identical element of the other model, in the meaning that all its attributes and all its references have also a mapping[3] in the other model.
For example, in our meta-model, the meta-element defining an attribute has a name, a reference on a type and a reference on its owner. Two attribute instances will be in full mapping if they have the same name, if their types are mapped and if their owners are also mapped.

Partial mapping an element has a mapping element in the other model in the meaning that an under-set of its attributes and references are also mapped[3] in the other model.
In our example, a class is in partial mapping with a class of the other model. Indeed, if the name must be the same one and if the list of attributes must be mapped for the two classes, we must not on the other hand check that their methods or their implemented interfaces are mapped. These elements belong to the model parts that are modified during the refinement application and their validity is checked separately.

[3] Mappings are thus checked in a transitive way (the mapping of the classes implies the mapping of the attributes of these classes which implies in its turn the mapping of the types of each attribute, etc.) but it is not necessary to remain in the same mode of mapping (full everywhere or partial everywhere). For example, the mapping of the classes could be full but the mapping of its attributes will be partial.

No mapping it is not necessary to check that an element has a mapping in the other model.
In our example, it should not be checked that an interface is in mapping with an interface of the other model. Indeed, during refinement, the interface list may change according to the designer choices (interface renaming, creation or removing, modification of the method list).

These mappings between elements are implemented in OCL by a series of utility functions (via the `def` operator). Depending on the complexity of the meta-model and the number of required mappings between the elements, the number of these functions can be important and tiresome to write by hand. This problem is easily avoidable through a dedicated tool able to automatically generate these functions for a given meta-model. Indeed, these functions are always based on the same structure, by checking the mapping of attributes and reference of an element in a transitive way. For a reference or attribute list, the mapping of the list elements is checked one by one. We have implemented a tool applying these principles. It analyzes any meta-model and proposes to the designer to choose, for each meta-model element type, the kind of desired mapping (for a partial mapping, the required attributes and references will be selected). The tool then generates all OCL mapping functions that the contract designer will complete or slighty modify for defining the specific part of the contract. Figure 3 is a screenshot of our tool in the context of mapping choices for the class element.

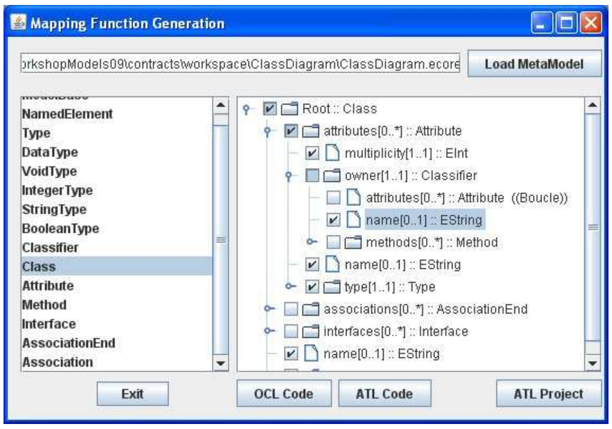

Fig. 3. Mapping function generation tool: selection for class mapping

When defining these functions, it is necessary to pay attention to an important detail: to avoid mapping cycles. For example, an attribute has a field `owner` of `Classifier` type referencing the owner of the attribute. When the mapping of two classes is checked, the mapping of their attributes is transitively checked; but if for an attribute we check the mapping of the `owner` field, we loop back in the checking of the mapping of the initial classes. The mapping of this field must then not be checked. If we need to have an in-depth mapping, i.e. applying transitively to a great number of elements, this problem becomes all the more important and complex to manage. Our tool informs the designer if such a cycle is present for the current mapping choices.

Constraints on element evolution for the interface addition example The third part of the contract, concerning the evolution of the elements between the source model and the target model is expressed on Figure 4 by the invariant of lines 10 and 11. It checks the contents of the source model and of the target models through the global model. `targetModel` and `sourceModel` corresponds, respectively to the model base of the source and the target model. They are easily defined through OCL in the following way, thanks to the `modelName` attribute added to each model element during the concatenation:

```
context ModelBase def: sourceModel : ModelBase =
                ModelBase.allInstances() -> any (modelName = 'source')
context ModelBase def: targetModel : ModelBase =
                ModelBase.allInstances() -> any (modelName = 'target')
```

The invariant consists in calling the function `sameClasses` (1) which, for two model bases, begins to check that the number of classes is the same (2). Then, for each class of the first model (3), it gets its set of methods by concatenating the methods of the class and of all its interfaces (4). It checks via the function `hasMappingClass` that the current class has a mapping in the other model (5) and if this is not the case the contract is not validated (9). It gets via the function `getMappedClass` the mapped class on the other model (6) and its complete set of methods (7). Then it is checked that the two classes have the same sets of methods via the call of `sameMethodSet` (8).

Due to lack of place, we will not detail all the mapping functions used by this contract. Those which relate to the classes are nevertheless defined on Figure 5. The main function is `classMapping` which applies a partial mapping between two classes: comparison of the name and of the attribute set (via the call of `sameAttributes`). The mapping functions for attributes will have a similar form.

The function `sameMethodSet` (Figure 6) compares two sets of methods. This is done by checking initially that the sets have the same number of elements. Then, it is checked, via the mapping function for method (`methodMapping`), that each element of the first set has a mapping element in the second one. Contrary to constraints of Figure 4 that have to be written explicitely by the contract designer, mapping functions of Figure 5 and Figure 6 are fully generated by our tool we described above.

We can notice that this contract not only checks that method sets are conserved for each class, but it also ensures that some elements are not modified

```
context ModelBase def: sameClasses(mb : ModelBase) : Boolean =
    self.allClasses -> size() = mb.allClasses -> size() and
    self.allClasses -> forAll( c |
        let myMethods : Set(Method) = c.interfaces -> collect(i | i.methods)
                                        -> union(c.methods) -> flatten() in
        if c.hasMappingClass(mb)
        then
            let eqClass : Class = c.getMappedClass(mb) in
            let eqClassMethods : Set(Method) = eqClass.interfaces -> collect(i|
                        i.methods) -> union(eqClass.methods) -> flatten() in
            c.sameMethodSet(myMethods, eqClassMethods)
        else
            false
        endif)

context ModelBase inv checkInterfaceContract:
targetModel.sameClasses(sourceModel)
```

Fig. 4. Evolution constraints between the source and target models

```
context Class def: classMapping(cl : Class) : Boolean =
    self.name = cl.name and
    self.sameAttributes(cl)

context Class def: hasMappingClass(mb : ModelBase) : Boolean =
    mb.allClasses -> exists( cl | self.classMapping(cl))

context Class def: getMappedClass(mb : ModelBase) : Class =
    mb.allClasses -> any ( cl | self.classMapping(cl))
```

Fig. 5. Mapping functions for a class

```
context Classifier def: sameMethodSet(
                    mets1 : Set(Method), mets2 : Set(Method)) : Boolean =
    mets1 -> size() = mets2 -> size() and
    mets1 -> forAll ( m1 |
        mets2 -> exists ( m2 | m1.methodMapping(m2)) )
```

Fig. 6. Mapping function for a method set

during the transformation. Indeed, the contract is validated if all classes are conserved with the same name and with the same attribute list. Il also verifies that all methods are conserved without modification of their signatures (through the calls of `methodMapping` in `sameMethodSet`). This ensures a well defined scope for the manual modification of the model.

All these functions and invariants forming the contract are written in standard OCL. It is thus checkable by any OCL evaluator implementing the standard and able to read Ecore models. From a practical point of view, we have chosen the ATL[4] tool to validate the contract on a global model. Although being basically a tool of model transformation, ATL fully implements the OCL standard and can easily be used to verify OCL constraints through a model transformation, as explained in [3].

Interest of explicit mapping functions Intuitively, one can consider that the problem of element mappings between the source and the target models is a key issue when it deals with expressing that such element (or such set of elements) corresponds in the other model to such other element (or such other set of elements) that can be very different in contents and structure. Indeed, this leads to complex mapping functions to define. These kinds of mappings are required in exogenous transformations but can also be defined for endogenous transformations. For instance, the OCL helper "sameClasses" of Figure 4 can be considered as a complex mapping function.

Our generated mapping functions only check that an element of a given type has an equivalent element of the same type on the other model. These mappings are then more simple to define. Nevertheless, we argue that they are key points in contract definition, and that they offer two major interests.

Firstly, it is important for an element to be able to get precisely its equivalent element on the other model, in order to apply evolution constraints to them. If we want to avoid mapping errors, we often need to express in-depth mapping, by transitively checking mappings of element attributes and references and so on. As an example, let consider our class diagram meta-model and the way it defines associations between classes. As we can have on a class diagram two associations with the same name, it is not enough to simply compare association names. We need to transitively compare their two association ends (name and bounds) and also compare the associated classes (at least their names) for these association ends.

Secondly, for almost all endogenous transformations and particulary refinements, we need to check that some model elements are not changed during the transformation. This is for the most important if we let the designer manually modify the model. For our example, we must check that name and attributes of classes and all associations are not modified during the interface addition refinement application. A contract dedicated to verify the unmodified parts of a model is only composed of mapping functions that can be entirely automatically generated. As a consequence, a contract validating unmodification on any

[4] Atlas Transformation Language: `http://www.eclipse.org/m2m/atl`

kind of model during an endogenous transformation can be fully automatically generated thanks to our mapping function generation tool.

5 OCL choice and related works

We have chosen to use standard OCL as contract expression language. We argue that this choice is relevant for several reasons. First, OCL is by nature a constraint expression language and a contract is a set of constraints. Next, our concern is to be able to define a contract, as far as possible, independently of modeling platforms and tools. In this scope, OCL is a good candidate since it is a standard usable on varied kinds of models, like UML, MOF or for the Eclipse/EMF platform. We can thus check OCL constraints on models generated by a wide range of transformation engines. Finally, OCL is a relatively well-known language. It is indeed difficult to define a precise meta-model or any model without the obligation to add assertions. For simplicity and commodity, these are often written in OCL. OCL is also used as a query language and is sometimes enhanced in transformation languages like ATL or QVT-compliant [12] languages like SmartQVT[5]. The last reason of choosing OCL is that it is a formal constraint language relatively well accepted by "lambda" designers which usually do not want to use formal techniques or languages. [13] argues also in this direction by writing that "(formal languages) are usually hardly accepted by software engineers".

Some other works have also chosen to use standard OCL to specify either model transformations or refinements, or even transformation contracts similar to ours. For example, [13] formally defines model refinements in OCL. [14] defines in OCL transformation contracts for ensuring model consistency and also use OCL for expressing code refactoring. [1,9] also propose to use model transformation contracts written in OCL to specify a transformation test oracle.

In almost all these approaches, as in ours, it is required to define mappings between elements of the source and the target models. But these mappings are always defined in an *ad hoc* way for the considered context, and sometimes, only in an implicit way. In contrast, we proposed a general method and a tool to explicitly define and generate mappings between elements in the context of endogenous transformations. The other difference is that most of these approaches are less general than ours on the way of obtaining and manipulating the models. Moreover, they can be dedicated to particular software environments and specific purpose. For example, the refinement specification in [13] is done by using the UML dependency relation stereotyped by `<<refine>>` which implies the definition of a UML diagram in which elements are explicitly bound by means of this dependency. However, no method or tool is proposed for, starting from two models (obtained in an unspecified way), automatically defining a model in conformity with this representation.

On another hand, model transformation verification is a large field where, of course, a lot of different techniques and tools can be used instead of OCL.

[5] `http://smartqvt.elibel.tm.fr`

These techniques can be based on formal specification, such as graph transformation and specification like in [10,15]. This presents the main drawback to be hardly accessible to the lambda designer. Avoiding this problem, recently dedicated MDE platforms or tools have introduced specification languages that can be used to verify transformations. We can cite for instance Kermeta[6], Epsilon[7] or openArchitectureWare[8]. Their languages are similar to OCL from the point of view of their goal, syntax (even if generally more simple), semantics and expressivity but with the possibility of manipulating several models simultanously. They are then good candidates in replacing OCL for expressing contracts but they oblige in learning and using a new language. So, from our point of view, the best solution will be simply to extend OCL to allow multiple contexts for OCL expressions, as for instance proposed by [8]. We say this is a need in order to improve the usability of OCL and should be included in the next OCL standard specification. Other secondary improvements, based on features available in the specification languages we just cited, would also be interesting, such as syntax simplification or outputs on constraint evaluation results. Finally, if remaining in the context of using standards, QVT Relation [12] would be a good candidate to express transformation contracts instead of OCL.

6 Conclusion and perspectives

We presented a method to specify and verify a model transformation contract concerning any kind of endogenous transformation. The general idea is to consider a couple of models, one being the source and the other the target of a transformation and to check that this couple strictly respects the transformation contract expressed as sets of invariants. The model transformation can be realized automatically and/or via a manual intervention of a designer. The contract is written in OCL to be as far as possible independent of platforms and tools.

We showed that the intuitive way of specifying a transformation contract through a couple of pre and post-conditions is not usable from a practical point of view, because it does not allow manual intervention on models during the transformation. We also showed the need and interest of detailing one-to-one mapping functions between elements. Indeed, they help in defining and structuring a contract. Moreover, they allow the direct and complete definition of a contract verifying unmodified parts of a model during its transformation. In the context of the Eclipse/EMF platform, we developped a tool that generates these mapping functions for any meta-model. For other technical spaces (UML, MOF, ...) similar tools could be defined to help in contract definition.

We pointed out the main problem of OCL, the single expression context, which leads to the need for concatenating two models within a more global model (this concatenation is realized automatically by our tools). Within model

[6] `htpp://www.kermeta.org`
[7] `htpp://www.eclipse.org/gmt/epsilon/`
[8] `htpp://www.openarchitectureware.org`

transformations, *i.e.* when manipulating several models at the same time, this becomes a very strong limitation and the next OCL specification should include a multi-context feature.

A forthcoming issue of our approach is to generalize it with exogenous transformations, *i.e.* with different source and target meta-models. The main difficulty is here too to go on the main limitation of OCL: constraints must only be written for a single meta-model context. A possible solution to this problem is for example to make the concatenation of the two meta-models in a more global third one in order to get a single meta-model as context for expressing OCL constraints, as suggested in [1].

References

1. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model Transformation Testing Challenges. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, July 2006.
2. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999.
3. J. Bézivin and F. Jouault. Using ATL for Checking Models. In *GraMoT 2005 workshop*, volume 152 of *ENTCS*, 2005.
4. E. Cariou, N. Belloir, and F. Barbier. Contrats de transformations pour la validation de raffinement de modèles *(in french).* In 5$^{\text{èmes}}$ *journées sur l'Ingénierie Dirigée par les Modèles (IDM 09)*, 2009.
5. 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.
6. E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the Specification of Model Transformation Contracts. Workshop OCL and Model Driven Engineering, UML 2004, 2004.
7. B. Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, 1992.
8. T. Milan, L. Sabatier, P. Bazex, and C. Percebois. NEPTUNE II, une plate-forme pour la vérification et la transformation de modèles. *Génie Logiciel*, (85), 2008.
9. J.-M. Mottu, B. Baudry, and Y. Le Traon. Reusable MDA Components: A Testing-for-Trust Approach. In *MoDELS 2006*, volume 4199 of *LNCS*. Springer Verlag.
10. A. Narayanan and G. Karsai. Towards Verifying Model Transformations. *ENTCS, Elsevier Science Publishers B. V.*, 211, 2008.
11. OMG. Object Constraint Language (OCL) Specification, version 2.0, 2006. `http://www.omg.org/spec/OCL/2.0/`.
12. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.0, 2008. `http://www.omg.org/spec/QVT/1.0/`.
13. C. Pons and D. Garcia. An OCL-Based Technique for Specifying and Verifying Refinement-Oriented Transformations in MDE. In *MoDELS 2006*, volume 4199 of *LNCS*. Springer Verlag, 2006.
14. P. Van Gorp. *Model-Driven Development of Model Transformations.* PhD thesis, University of Antwerp, Dept. of Mathematics and Computer Science, 2008.
15. D. Varro. Towards Formal Verification Of Model Transformations. In *PhD Student Workshop of FMOODS 2002*, 2002.