A Black-Box and Contract-Based Verification of Model Transformations

Meriem Lahrouni[1,2], Eric Cariou[2], and Abdelaziz El Fazziki[1]
[1]Computer Science Department, University Cadi Ayyad, Morocco
[2]Computer Science Laboratory, University of Pau and Pays de l'Adour, France

Abstract: *The main goal of Model-Driven Engineering (MDE) is to manipulate productive models to build software. In this context, model transformation is a common way to automatically manipulate models. It is then required to ensure that transformation has been correctly processed. In this paper, we propose a contract-based method to verify that a target model is a valid result of a source model with respect to the transformation specification. The verification is made in a black-box mode, independently of the implementation and the execution of the transformation. The method allows the contract to be written in any constraint language. In association with this method, we have implemented a tool that partially generates contracts written in OCL and manages their evaluation for both endogenous and exogenous transformations.*

Keywords: *MDE, model transformation, contract, verification.*

Received October 26, 2015; accepted July 19, 2017

1. Introduction

The main goal of Model-Driven Engineering (MDE) is to manipulate productive models to build software. MDE-based software is being more and more prevalent since it can be applied to all domains for different purposes. For example, Nouzri and Fazziki [15] proposed a methodology that could make the development of complex information systems better aligned, easier and less costly, and Tounsi *et al.* [19] defined a MDE-based approach for the simulation of a supply chain. In this context, models must be precisely defined for being automatically manipulated by tools. The most common manipulation of models is a model transformation where a target model is generated based on a source model[1]. Model transformations are implemented through dedicated languages and, as for any programming task, are subject to errors or implementation bugs. For this reason, a lot of works focus on verifying or validating model transformations. For instance, Rahim and Whittle [17] have studied no less than 57 model transformation verification approaches. Verification is based on contracts, testing, model-checking or theorem proving. Boehm [3] defines verification as building the thing right and validation as building the right thing. In other words, verification consists in ensuring that a software artifact respects its specification and validation assures that this specification is the expected one.

Programming and design by contract are well-known lightweight verification approaches [1, 12, 13]. In [5, 6, 7], we have applied the principles of contracts to the context of model transformation, defining in this way model transformation contracts. Contracts aim at ensuring that a target model (the model after the transformation) is valid regarding a source model (the model before the transformation).

In this paper, we extend our previous works on contracts by proposing a framework and a tool[2] for implementing model transformation contracts. The tool has been developed for Ecore metamodels and is using by default Object Constraint Language (OCL) [16] for the contract implementation. The verification is made in a black-box mode and has been designed for being open and independent. We ensure the following properties:

- The verification is carried out independently of the transformation execution and implementation.
- Both endogenous and exogenous transformations can be verified. Endogenous transformations are transformations between models expressed in the same modeling language and exogenous transformations are transformations between models expressed in different languages.
- The evaluation of the contract is exploitable: in case of problems, the model elements that do not respect their part of the contract are clearly identified.

White-box verification is strongly linked with the implementation or the execution of the transformation.

[1] All the explanations of this paper are based on a single source model and a single target model of a transformation. However, the presented approach and the associated tool are easily and directly generalizable to handle several source and target models.

[2] The contract tool and implementation of examples presented in this paper are available online: http://web.univ-pau.fr/ecariou/iajit/

It cannot then be used for verifying manual transformations. On the other side, black-box verification offers a wider scope of verification for a couple of models (source and target models of a transformation). For instance, it can ensure that models manually modified by the designer respect the excepted constraints.

The rest of this paper is organized as follows. The next section introduces the principles of model transformation contracts and presents two transformation examples to illustrate our approach. Section 3 presents the framework of contract definition ensuring the above properties and how models and metamodels are handled in this context. Section 4 describes the mappings of elements between a source and a target model. In our context, mappings define how considering equivalent elements of different models. They are required for expressing constraints on the evolution of the elements through the transformation. Finally, related work is discussed before concluding.

2. Model Transformation Contracts

In this section, we first introduce the principles of model transformation contracts and then, two examples of model transformations and their associated contracts are presented.

2.1. Definition of Model Transformation Contracts

Programming and design by contract consist in specifying what a software component, a program or a model does, in order to know how to properly use it. Design by contract also allows at runtime the assessment of what has been computed with respect to the expressed contracts. A contract is composed of two kinds of constraints:

- Invariants that have to be respected by software elements.
- Specification of operations on the software elements through pre and post-conditions. A precondition defines the state of a system to be respected before its associated operation can be called in a safe mode. Post-conditions establish the state of a system to respect after calls. If a pre-condition is violated, post-conditions are not ensured and the system can be in an abnormal state.

In the MDE context, a metamodel is a structural diagram defining the types of model elements and their relationships. But this structural view is rarely sufficient for expressing all relations among elements; we need to complement it with well-formedness rules, which are additional constraints expressed in a dedicated language such as OCL. Contract invariants can be typically rules or any supplementary constraints. Operations specified through a contract could be any kind of model manipulation and modification, such as model transformations.

In [7], an approach for specifying contracts on model transformation operations using OCL has been proposed. These contracts describe expected model transformation behavior. Formally, constraints on the state of a source model are offered. Similar constraints on the state of the target model are offered as well. 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 couple of pre- and post-conditions for specifying a transformation can also be organized via three distinct sets of constraints:

- *Constraints on the source model*: constraints that a model must respect for being able to be transformed;
- *Constraints on the target model*: general constraints (independent of the source model) to be respected by a model for being a valid result of the transformation;
- *Constraints on element evolution*: constraints on the evolution of elements between the source and the target models. They ensure that the target model is the correct transformation result according to the source model content.

2.2. Examples of Model Transformations

As an illustration, we describe two examples of transformations and their associated contracts. The first transformation is endogenous whereas the second is exogenous. These transformations are based on a basic class diagram metamodel and on a database metamodel that are first described.

2.2.1. A Basic Class Diagram and a Database Schema Metamodels

The basic class diagram metamodel is shown in Figure 1, left part. A class diagram consists of classes, associations and data types (String Type, Integer Type and Boolean Type). A class contains zero or more attributes and can specialize other classes. An attribute type is a data type. An association between classes is defined by two ends. Each end has a lower bound, an upper bound and is associated with one class.

The metamodel for the database schema models is shown in Figure 1, right part. A relational schema consists of tables and a set of types (IntType, VarcharType and BoolType). A table consists of zero or more columns, keys and foreign keys. Some of these columns can be included in a key to indicate that the column forms a part of the table's key. Each foreign key refers to the key of the table it identifies, and indicates one or more columns in the table as being part of the foreign key. Each column is typed.

These metamodels are augmented with OCL invariants for expressing the well-formedness rules. For instance for class diagrams, there is no cycle in the specialization of classes and classes have a unique name. Due to lack of space, they are not presented.

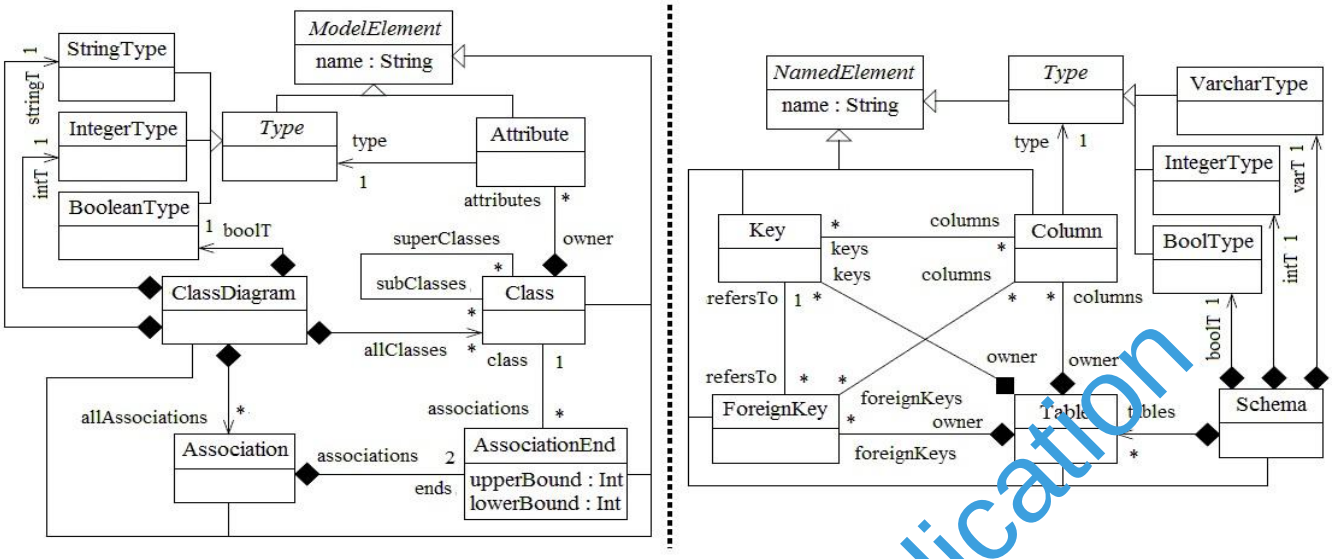

Figure1. Class diagram and database schema metamodels.

2.2.2. Removing Super-Classes

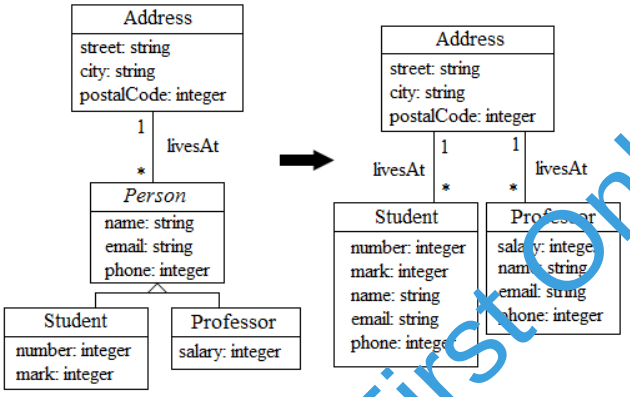

Figure 2. Example of super-classes removing.

An example of endogenous transformation is a class diagram refactoring. It consists in removing all the inheritance links between classes. For that, it duplicates the attributes of a super-class into its leaf sub-classes. In the same way, associations coming from super-classes are moved to their leaf sub-classes. Figure 2 gives an example of such refactoring. One can notice that the Person super-class has been removed and that all its attributes have been duplicated in the Student and Professor sub-classes. The association livesAt is also duplicated for both subclasses.

The contract associated with this transformation is the following:

- *Constraints on the source model*: none, any class diagram can be transformed.
- *Constraints on the target model*: no super-class for any class.
- *Constraints on element evolution from the source model towards the target model:* all classes without sub-classes are maintained, others are removed. Each remaining class has the same attribute (resp. association) set augmented with the attributes (resp. associations) of its super-classes.

2.2.3. From Classes to Database Tables

The exogenous transformation is the classic example of translation of a class diagram to a relational database schema: each class becomes a table with its primary key, each attribute becomes a column of a table and each association is transformed to foreign keys in the associated tables. For instance, the class Professor of Figure 2, right side, leads to a table definition Professor(int professor_id, int address_fk, int salary, varchar (40) name, varchar (40) email, int phone) with professor_id the primary key of the table and address_fk is a foreign key referencing the Address table.

The contract associated with this transformation is the following:

- *Constraints on the source model*: no super-class for any class.
- *Constraints on the target model*: none.
- *Constraints on element evolution from the source model towards the target model*: each source element has its corresponding target element according to the transformation correspondences and keys are generated for tables.

3. Model and Metamodel Management

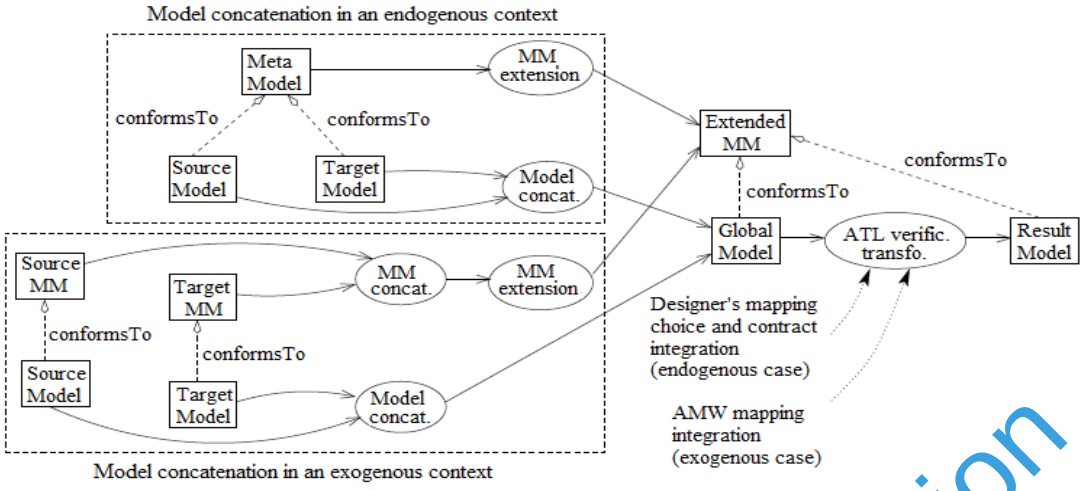

Figure 3. Contract definition and evaluation process.

As stated in the introduction, we want a contract to be implementable in any language. It can be based on a common programming language or, in a more suitable way, on a constraint language. The most common MDE-related constraint language is OCL but there are other ones such as EVL[3]. The problem is the expression of constraints on the element evolution during the transformation. Indeed, it implies expressing constraints simultaneously on elements of both source and target models. If EVL can express constraints for several models at the same time, this is not the case for OCL. Therefore, in order to make our approach as open as possible, we must rely on the most restricted context. Concretely, OCL implies the evaluation of constraints on a single model. As we need to express constraints on two models, the solution is then to concatenate these models into a single one. This later is made conforming to a global metamodel (the result of concatenation of source and target metamodels) that contains meta-elements that allow the contract evaluation. The concatenation is made automatically by the tool and constitutes the key of our transformation verification approach. Indeed, it gathers all source and target elements in one global model that conforms to an extended metamodel allowing thus the contract evaluation, which will be explained in this section.

3.1. The Contract Evaluation Process

Figure 3 shows the process enabling the definition and the evaluation of a contract, based on automatic manipulations of models and metamodels. The first part of the process consists in modifying the metamodel(s) for processing the automatic concatenation of the source and the target models into a global model. This is achieved in different ways depending on the endogenous or exogenous nature of the transformation. A contract written in any language can then be evaluated on the global model. We propose to integrate the result of this evaluation directly in the global model leading to a result model.

In the context of OCL as the contract language, we use a partially generated ATLAS Transformation Language (ATL) transformation for evaluating the contract. This transformation integrates the designer's constraints forming the contract and its mapping choices (enabling correspondences between the source and the target model elements).

The rest of the section details the points of this process, except the mappings which are defined in the next section.

3.2. Metamodel and Model Management for Endogenous Transformations

In the context of an endogenous transformation, concatenating the source and the target models is technically simple. However, we need to keep a trace of the origin of each element in the global model. For instance, when concatenating the two models shown in Figure 2, the global model contains two classes named "Student" and it is important to know which one is coming from the source model and which one from the target model. To achieve this, the contract tool realizes an automatic extension of the metamodel without modifying its original elements. A model conforming to a metamodel will also directly conform to its extended version. This extension adds into each meta-class an attribute called "modelName" and used for tagging each element of the global model with a "source" or "target" value.

To generate the global model, the tool takes as input source and target models as well as the extended metamodel. It adds all elements of the source model and all elements of the target model into a third global model conforming to the extended metamodel. During

[3] Epsilon Validation Language: https://www.eclipse.org/epsilon/doc/evl/

this step, each element is tagged with "source" or "target" string value[4], depending on the model it belongs to. As output, our tool returns the global model containing all elements of both source and target models with indication of their origin.

3.3. Metamodel and Model Management for Exogenous Transformation

In the context of an exogenous transformation, the metamodel extension is not sufficient. The issue is that the elements of the global model are conforming either to the source metamodel or to the target metamodel. As it is not possible for a model to conform to two metamodels at the same time, the solution is to create a metamodel to which all the elements of either the source or the target model can conform. This is achieved by concatenating all the meta-elements of the source and the target metamodel within a third global metamodel. This can however lead to a problem if two meta-elements have the same name in each metamodel. To avoid this problem, the tool renames all the meta-elements with a prefix "S_" or "T_" that indicates whether the meta-element comes from the source or the target metamodel. For example, if a transformation takes as source model a class diagram, the global metamodel will contain S_Association meta-element that is the renaming of Association meta-element of the class diagram metamodel.

In addition to the metamodel concatenation, the tool extends also the obtained global metamodel in the same way as for an endogenous transformation. When the source and the target models are concatenated, their elements are tagged and their instantiation links are modified. For instance, if we consider the class diagram of Figure 2, right part, as a source model of an exogenous transformation, the instance of Class named "Student" will become in the global model an instance of S_Class named "Student" and tagged as model name with the "source" string value.

3.4. Contract Implementation and Evaluation

The evaluation of the contract written in any language can be processed once source and target models are concatenated through our tool. In order to facilitate the interpretation of the result of contract evaluation, we propose to integrate it directly within the concatenated model. For that, the metamodel extension defines a set of meta-elements for expressing the result of the contract evaluation: ContractError, ContractWarning and ContractCorrect. Each one contains a comment and references an element of the global model. This enables to precisely specifying for each element, either from the source or the target model, if it is respecting or not its part of the contract. Details concerning the metamodel extension and the added elements can be found in our previous work [8].

```
helper context MM!Class def : ClassContractInvariant() : Boolean =
  true;
abstract rule duplicateClass {
  from
        sourceClass : MM!Class
  to
        targetClass : MM!Class (
              -- Duplication of Class elements)}
rule duplicateClassOk extends duplicateClass {
  from
        sourceClass : MM!Class (
              sourceClass.ClassContractInvariant())
  to
        targetClass : MM!Class ( ) }
rule duplicateClassNOk extends duplicateClass {
  from
        sourceClass : MM!Class (
              not sourceClass.ClassContractInvariant())
  to
        targetClass : MM!Class ( ), error : MM!ContractError (
              comment <- 'The class element [
               modelName ='+sourceClass.modelName.toString()
               +' name ='+sourceClass.name.toString()+' ]
              is not respecting its contract part', element <- sourceClass) }
```

Figure 4. ATL code generation for the meta-element Class.

Moreover, a transformation operation can take parameters and the contract can integrate these parameters in its definition. Elements of the global model are referenced for specifying parameters (as well as the return value of operations). These elements are not necessarily elements of the source or the target model, but they can be additional elements. In this case, they have to be tagged with a string value different from "source" or "target".

Finally, the tool can generate a skeleton of contract implementation with OCL as contract language. For that, it generates an endogenous ATL[5] transformation. This verification transformation takes as input the global model and generates the result model that contains the contract evaluation results. ATL is a transformation language based on OCL. It can be used to define and evaluate OCL constraints on models [2] and then to define an OCL contract. For each meta-element of the metamodel which conforms to the global model, the tool generates an empty OCL helper of Boolean type and a couple of transformation rules (Figure 4). These rules fully duplicate the element content, but in case of non-respect of the contract, an error message that references the current element is generated in addition. This generic comment can of course be modified by the designer to express a more accurate error message. The idea is that the OCL helper, that returns a boolean, will contain the part of the contract for this kind of meta-element.

All the three types of constraints can be directly implemented within these helpers. For example, any

[4] In case of multiple source and multiple target models, it is necessary to differentiate between source models in one hand and target models in the other hand. Element are then tagged with "source1", "source2"..., "target1", "target2"... string values, depending on the order of the model they belong to.

[5] http://www.eclipse.org/atl/

instance of Class from the target model must have an empty set of super-classes, that is, does not have any super-class. This constraint on the target models of the class diagram refactoring is simply expressed as shown in Figure 8 (line 7).

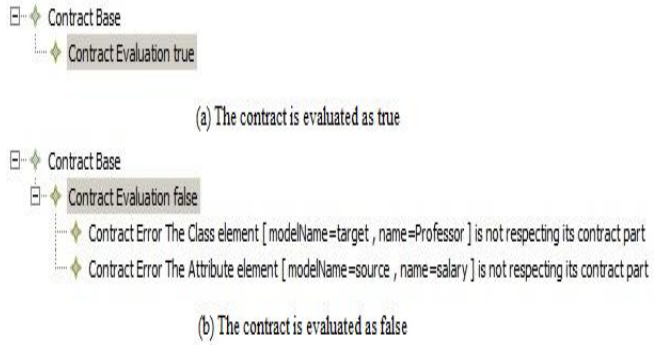

Figure 5. The contract evaluation.

Figure 5 shows two screenshots of the result model in the two cases. The contract is evaluated as true in the top part of the figure. After removing the attribute salary from the target class Professor, the contract was evaluated as false for the class Professor because it does not contain all its previous attributes and for the salary attribute because it has no equivalent in the target model. This self-contained result model can then directly be read by a tool aiming at presenting the contract evaluation.

4. Mapping for Element Evolution Specification

Constraints on elements of source and target models are easy to implement as seen in the previous section. However, constraints on the evolution of elements between the source and the target models raise a problem. With a standard operation specification with pre and post-conditions, it is possible to reference in the post-condition of a transformation operation both elements of the source and of the target models, thanks to the @pre construction in OCL. However, this requires to verify the transformation only during its execution and to implement the contract jointly with the transformation. These requirements are incompatible with our choice of black-box verification. They also prevent the verification of target models that have been manually modified by the designer.

Expressing the evolution between source and target elements is based on mappings that allow finding, for each element of the source –(resp. target) model, its corresponding element in the target (resp. source) model. Mapping functions are defined as relationships between source and target elements and are implemented through OCL helpers. Mappings can be defined in endogenous and exogenous contexts. Our tool helps the contract designer by generating the OCL helpers based on his mapping choices.

4.1. Endogenous Mappings

Endogenous mappings and their implementation through the first version of our contract tool have been presented in [7]. The current implementation of the tool has mainly enhanced the management of associations between elements depending on the association properties (unique, ordered, etc.). Below is an introduction and example of endogenous mappings.

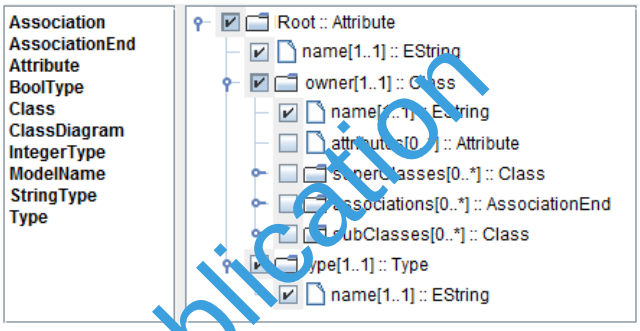

Figure 6. Screenshot of the contract tool in the context of endogenous mappings selection.

Endogenous mappings aim at finding, within the global model, an element from a model (source or target) that has a corresponding element of the same type in the other model. For instance, for verifying that the Student class has after the transformation the right set of attributes, it is required to first obtain the Student class of the source model so that expected attributes could be obtained. These two elements are from the same type (the Class meta-element) but one is tagged “source” and the other one “target”. In addition, the designer needs to select, for each required meta-element, the attributes and references on the other related meta-element and their contents that make sure, with equality of their values, that the two elements are mapped. For classes, it simply consists of comparing their names as they must be unique but this is not as simple for other meta-elements.

Figure 6 is a screenshot of our contract tool in the context of endogenous mapping selection. The left part lists all the meta-elements and the right part allows the designer to make his mappings for each meta-element. The figure shows the mappings selected for the meta-element Attribute. The tool displays, in the form of a tree, all features of the meta-element with their types and cardinalities. The designer can select mapping criteria for each meta-element by simply checking some features. A selected meta-element feature has to remain with the same value both in source and target models. Criteria in the example mean that two attributes are considered equivalent if they have the same name and type and

belong to the same class. The same class and same type are defined as comparing their respective names.

```
1  helper context MM!Attribute def : hasMappingOnOtherSide_Attribute() : Boolean =
2          if self.modelName = 'source'
3          then MM!Attribute.allInstances() -> exists ( a |
4                  a.modelName = 'target' and self.mappingAttribute(a) )
5          else MM!Attribute.allInstances() -> exists ( a |
6                  a.modelName = 'source' and self.mappingAttribute(a))
7          endif;
8  ----------------------------------------------------------------------------
9  helper context MM!Attribute def : getMappedOnOtherSide_Attribute() : Boolean=
10         if self.modelName = 'source'
11         then MM!Attribute.allInstances() -> any ( a |
12                 a.modelName = 'target' and self.mappingAttribute(a) )
13         else MM!Attribute.allInstances() -> any ( a |
14                 a.modelName = 'source' and self.mappingAttribute(a))
15         endif;
16 ----------------------------------------------------------------------------
17 helper context MM!Attribute def : mappingAttribute ( attribute : MM!Attribute ) : Boolean=
18     self.name=attribute.name and
19     self.owner.mappingAttribute_owner( attribute.owner ) and
20     self.type.mappingAttribute_type( attribute.type );
21 ----------------------------------------------------------------------------
22 helper context MM!Class def : mappingAttribute_owner ( class : MM!Class ) : Boolean =
23     self.name=class.name;
24 ----------------------------------------------------------------------------
25 helper context MM!Type def : mappingAttribute_type ( type : MM!Type ) : Boolean=
26     self.name=type.name;
```

Figure 7. Mapping functions generated for the meta-element attribute.

```
1  helper context MM!Class def : ClassContractInvariant() : Boolean =
2          self.modelName = 'source' implies (
3                  if self.subClasses -> notEmpty() then not self.hasMappingOnOtherSide_Class()
4                  else self.hasMappingOnOtherSide_Class()
5                  endif) and
6          self.modelName = 'target' implies (
7                  self.superClasses -> isEmpty() and
8                  if not self.hasMappingOnOtherSide_Class() then false
9                  else self.consistentAttributesAndAssociations(self.getMappedOnOtherSide_Class())
10                 endif);
11 ----------------------------------------------------------------------------
12 helper context MM!Class def: consistentAttributesAndAssociations(class : MM!Class) : Boolean=
13                 self.hasPreviousAttributes(class) and
14                 self.hasSuperClassesAttributes(class) and
15                 self.hasPreviousAssociations(class) and
16                 self.hasSuperClassesAssociations(class);
17 ----------------------------------------------------------------------------
18 helper context MM!Class def: hasPreviousAttributes(class : MM!Class) : Boolean =
19         let previousAttributes : Set(MM!Attribute) = MM!Attribute.allInstances() -> select(a | a.owner = class) in
20         previousAttributes -> forAll (a1 | self.attributes -> exists(a2 | a1.name = a2.name and a1.type.name =
21         a2.type.name));
```

Figure 8. Excerpt of the contract invariant for the meta-element Class and the class diagram refactoring.

Figure 7 presents the ATL code of the mapping functions generated for the meta-element Attribute based on the designer choice of Figure 6. The helpers hasMappingOnOtherSide_Attribute (line 1) and getMappedOnOtherSide_Attribute (line 9) aim at looking for or getting the attribute of the other model mapped with the current attribute. For that, the attribute search is made depending on the modelName value and based on the attribute mapping function. This helper is defined at line 17 and checks the equality of the attribute's name and of the mapping of their owners and types through the mapping helpers mappingAttribute_owner and mappingAttribute_type defined respectively at lines 22 and 25. They simply compare the names of classes and types.

```
1  helper context MM!S_Class def : mapping_S_Class_T_Table ( t : MM!T_Table) : Boolean =
2          self.name = t.name;
3  ----------------------------------------------------------------------------
4  helper context MM!S_Class def: hasMappingOnOtherSide_S_Class_T_Table() : Boolean =
5          MM!T_Table.allInstances() -> exists ( ins | self.mapping_S_Class_T_Table(ins));
6  ----------------------------------------------------------------------------
7  helper context MM!S_Class def : getMappedOnOtherSide_S_Class_T_Table() : MM!T_Table =
8          MM!T_Table.allInstances() -> any (t | self.mapping_S_Class(t));
```

Figure 9. Mapping functions generated for the relationship Class-Table.

Now, if we suppose that the designer has defined the mappings between classes (hasMappingOnOther-Side_Class/getMappedOnOtherSide_Class generated helpers that simply compare the class names), the contract invariant for classes can be completed as shown in Figure 8: A class on the source side (line 2) with sub-classes must be removed and then has no mapping on the target side (line 3). Otherwise, it is kept and then has a mapping (line 4). On the target side (line 6), a class has no super-class (line 7), must correspond to an existing class on the source side (line 8) and must have a consistent set of attributes and associations based on its mapped class on the source side (line 9). A target class must contain all its previous attributes (resp. associations) in addition to all the attributes (resp. associations) of its previous super-classes (lines 13 to 16)–due to lack of space, only the helper hasPreviousAttributes (line 18) is presented. It checks whether each attribute of the class passed as parameter has an equivalent attribute (with same name and type) in the current class.

4.2. Exogenous Mappings

Exogenous mappings consist in expressing correspondences between elements of different types that belong to different models (source or target models). Several tools exist for automatically generating mappings between models or metamodels based on the similarities of element contents [18]. Our tool is currently using the AMW matching[6] but could be easily extended to work with other matching tools or techniques. The tool takes an AMW weaving model to generate the mapping functions in the case of an exogenous model transformation. These mapping functions are generated within the ATL verification transformation that is used to evaluate the contract. The AMW weaving model can be obtained in several ways, either written by hand by the designer or based on an automatically defined one. Indeed, AMW generates automatically, by executing a series

[6] Atlas Model Weaver (AMW): http://www.eclipse.org/gmt/amw/

of heuristic algorithms, a weaving model that contains relationships between source and target metamodels. Produced relationships can be manually modified in order to get correct and consistent mappings.

For our exogenous example, a class of the class diagram metamodel is transformed to a table of the database metamodel with the same name. Figure 9 shows the generated mapping functions. Line 4, hasMappingOnOtherSide_S_Class_T_Tabe helper verifies that there is a target table mapping the source class by checking that these two elements have the same name through the mapping_S_Class_T_Table mapping function (line 1). getMappedOnOther-Side_S_Class_T_Table (line 7) returns this target table.

4.3. Mappings as Part of the Contract

The main goal of the mappings is to be able to get a corresponding element of a current one in order to express constraints between them. However, simply having or not having a mapping between source and target elements can also be directly a part of the contract. Indeed, in an endogenous case, no mapping can mean that the element has been removed and having an element of the same type with some identical values is a constraint on the evolution of the element content: some of its attributes and relations must not change. For our example, mappings are used to express that a class on the target side has its equivalent class in the source model (i.e., it is not created from nowhere) and that a source class with subclasses must not have a corresponding class on the target side as it must be removed by the transformation.

More generally, endogenous mappings can define constraints on unmodification of elements during the transformation. As a consequence, based on mapping choices, our tool can generate automatically, still under the form of an ATL verification transformation, an unmodification contract ensuring that some parts of the model are not modified during the transformation. Depending on the transformations, such unmodification verification of a part of a model can form an important part of the complete contract.

In an exogenous context, constraining the target element to have an equivalent source element of a different type indicates that this source element has been correctly transformed. For our exogenous example, mappings will ensure that a class has a corresponding table, an attribute has a corresponding column, etc. For this transformation, mappings form the major part of the contract.

5. Related Work

There are several surveys on the state of art of model transformation verification [4, 17]. Contract approaches are cited as one way of verifying model transformations. Part of the interests of contracts is that they can be used solely as a verification approach or as an oracle in model transformation testing.

There are several contract-based approaches in the context of model transformations. A lot of them have also chosen OCL for implementing the contracts. For example, authors in [10] define transformation contracts for the properties that need to be checked and uses them to check input test models automatically transformed into output models. Van Gorp defines in OCL transformation contracts for ensuring model consistency [20]. Mottu *et al*. [14] propose to use model transformation contracts written in OCL to specify a transformation test oracle. Almost all of these approaches are dedicated to particular software environments and for specific purposes. For example in [10], test models are checked using the USE tool [9], after an automatic transformation into output models. However, no method or tool is proposed, starting from two models (obtained in an unspecified way), for automatically defining a model in conformity with this representation. The other difference is that most of these approaches define mappings between elements of the source and the target models in an ad hoc way and sometimes only implicitly. For example, the designer writes manually mapping functions for the considered context. In contrast, we propose a general method and a tool that explicitly define and generate mappings between elements.

Guerra *et al*. [11] propose a model transformation contract approach in a black-box mode as we do. They go further by implementing a testing tool based on their contracts. Contracts are defined using a visual language making them easier to define than with a textual constraint language such as OCL, thus avoiding the necessity of model concatenation. However, the restriction is that the contract must be defined using their own language and verified by their tool.

6. Conclusions

In this paper, we present a contract-based black-box method to verify that a model transformation has been correctly carried out (including manual transformations), starting from a couple of models, one being the source and the other the target of a transformation. The approach has been designed to be more independent of tools and languages, either from the transformation implementation or the writing of the contract. For this purpose, we have developed a contract tool that processes manipulation and modification of models and metamodels for concatenating within a global model the source and the target models of a transformation. Indeed, some constraint languages, such as OCL, can only express constraints on a single model. For expressing constraints on the evolution of elements between the

source and the target models, these elements need to be within the same model. We then show the need and interest of mapping functions after criteria selection by the designer. Mappings help in writing a contract by defining equivalent elements between the source and the target models within the global model. Moreover, mappings are also part of the contract definition. The contract tool generates in the context of Ecore metamodels and OCL, an ATL transformation embedding the generated mappings and the contract defined by the designer. This transformation adds within the global model the result of the contract evaluation referencing precisely each element causing problem. Compared to the first version of the contract tool presented in [7], the tool can now manage exogenous transformations and generates this ATL evaluation transformation.

In the future, we plan to extend our tool to write contracts in other constraint languages such as EVL or to use other matching definition files in addition to AMW. The tool has also to be repackaged for being available as an Eclipse plugin. We also intend to use contracts for other purposes than model transformation. For example, we could use contracts written in Temporal Object Constraint Language (TOCL) [21] to specify constraints on the temporal evolution of model execution that is considered as a sequence of model transformations as explained in [8]. Contracts could also be applied to co-evolution in order to verify the respect of constraints on the evolution of a model following the evolution of its metamodel.

References

[1] Beugnard A. et al., "Making Components Contract Aware," *IEEE Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[2] Bézivin J. and Jouault F., "Using ATL for Checking Models," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 69-81, 2006.

[3] Boehm B., "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, vol. 1, no. 1, pp. 75–88, 1984.

[4] Calegari D. and Szasz N., "Verification of Model Transformations: A Survey of the State-of-the-Art," *Electr. Notes Theor. Comput. Sci.*, vol. 292, pp. 5–25, 2013.

[5] Cariou E. et al., "Model Transformation Contracts and their Definition in UML and OCL," *Techn. Ber.*, vol. 8, April 2004.

[6] Cariou E. et al., "OCL for the Specification of Model Transformation Contracts," *In Proceedings of Workshop OCL and Model Driven Engineering*, Lisbon, Portugal, 2004.

[7] Cariou E. et al., "OCL Contracts for the Verification of Model Transformations," *Electronic Communications of the EASST*, vol. 24, 2010.

[8] Cariou E. et al., "Contracts for model execution verification," *In Proceedings of European Conference on Modelling Foundations and Applications*, Birmingham, UK, pp. 3–18, 2011.

[9] Gogolla M., Büttner F., and Richters M., "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1, pp. 27-34, 2007.

[10] Gogolla M. and Vallecillo A., "Tractable Model Transformation Testing," *In Proceedings of European Conference on Modelling Foundations and Applications*, Birmingham, UK, pp. 221-235, 2011.

[11] Guerra E. et al., "Automated verification of model transformations based on visual contracts," *Automated Software Engineering*, vol. 20, no. 1, pp. 5–46, 2013.

[12] Le Traon Y., Baudry B., and Jézéquel J.-M., "Design by contract to improve software vigilance," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 571– 586, 2006.

[13] Meyer B., "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40-51, 1992.

[14] Mottu J.-M., Baudry B., and Le Traon Y., "Reusable MDA Components: A Testing-for-Trust Approach," *in Proceedings of International Conference on Model Driven Engineering Languages and Systems*, Genova, Italy, pp. 589-603, 2006.

[15] Nouzri S. and Fazziki A., "A Mapping from BPMN Model to JADEX Model," *The International Arab Journal of Information Technology*, vol. 12, no.1, pp. 77-85, January 2015.

[16] OMG, "Object Constraint Language (OCL) Specification, version 2.2," 2010, http://www.omg.org/spec/OCL/2.2/.

[17] Rahim L. A. and Whittle J., "A survey of approaches for verifying model transformations," *Software and System Modeling*, vol. 14, no. 2, pp. 1003-1028, 2013.

[18] Stephan M. and Cordy J. R., "A Survey of Methods and Applications of Model Comparison," *Queen's University, Tech. Rep*, vol. 582, pp. 2012, 2011.

[19] Tounsi J., Boissière J., and Habchi G., "Multiagent decision making for SME supply chain simulation," *In Proceedings of 23rd European Conference on Modeling and Simulation (ECMS)*, Madrid, Spain, pp. 203-211, 2009.

[20] Van Gorp P., "Model-Driven Development of Model Transformations," *In Proceedings of International Conference on Graph*

Transformation, Leicester, United Kingdom, pp. 517-519, 2008.

[21] Ziemann P. and Gogolla M., "OCL Extended with Temporal Logic," *In Conference on Perspectives of System Informatics*, Novosibirsk, Russia, pp. 617–633, 2003.

Meriem Lahrouni is a member in the research laboratory (Computer System Engineering) of the Computer Science Department at the Faculty of Semlalia (Marrakech, Morocco) for the preparation of her thesis; she worked on the contract-based verification of model transformations.

Eric Cariou is an associate professor of computer science at the University of Pau (France). His research interests include software architecture, model-driven engineering, contract-based verification of transformations, model execution, and software adaptation. Cariou received a PhD in computer science from the University of Rennes (France).

Abdelaziz El Fazziki is a Professor of computer science at Marrakech University, where he has been since 1985. He received an MS from the University of Nancy (France) in 1985. He received his PhD in computer science from the University of Marrakech in 2002. His research interests are in software engineering, focusing on information system development.