# Leveraging Model Transformations by means of Annotation Models

Juan M. Vara, Verónica A. Bollati, Belén Vela, Esperanza Marcos

Research Group Kybele
Rey Juan Carlos University
Madrid (Spain)
{juanmanuel.vara, veronica.bollati, belen.vela, esperanza.marcos}@urjc.es

**Abstract.** Model transformations are the key to automate any software development proposal based on model-driven engineering. However, it might happen that a *unique* transformation does not suit for every possible scenario. This could be the case when the gap between source and target metamodels is too large or the target metamodel is too complex. In such situations, it may happen that the transformation never generates some constructions, unless its execution is driven to do so. In other words, to obtain the most accurate models we need to introduce some design decisions that guide the transformation. A way to do so is to model our design decisions as annotations over the source model – in a model-driven engineering context, everything should be a model. Then, we can use such annotation model as an additional input for the model transformation. This work shows how we have applied that technique to improve our proposal for model-driven development of XML Schemas. The solution is based on the use of weaving models as annotation models.

**Key words**: Model-Driven Software Development, XML Schema, Annotation Models, Weaving Models.

## 1 Introduction

Since the World Wide Web Consortium (W3C) proposed the eXtensible Markup Language (XML), it has become the current *de facto* standard for information interchange between different organizations.

Initially, the way to define the structure of XML document was by declaring a Document Type Definition (DTD). DTDs were very efficient at the beginning. However, as the use of XML documents increased, the weaknesses of DTDs arose. They present syntactic and semantic failings, especially when the structure of the conforming XML documents is complex. For instance, they are not well-formed XML documents, thus developers have to learn how to use two different syntax. Besides, their mechanisms for defining arity are rather poor. To overcome these drawbacks, the W3C proposed a new standard for defining the structure of XML

documents: the XML Schema Language [23]. It is an alternative to the use of DTDs based on XML that provides a series of advantages with respect to DTDs.

The main improvement of XML Schemas regarding DTDs was providing with a vastly improved data typing system. XML Schemas also support namespaces, which allow different parts of a particular XML document to conform to different XML Schemas [2]. All this given, the XML Schema has been commonly adopted as the de-facto standard for XML document modeling.

In the line of the new trend in software development, in [4] we applied the principles of the Model-Driven Engineering (MDE) approach [20] to the development of XML Schemas. MDE proposes the use of models in each step of the development process. Such models represent the Information System (IS) at different abstraction levels. Besides, the transformation rules between these models have to be defined.

Our proposal starts from a Platform Independent Model (PIM) represented by a UML class diagram. Next, a model to model transformation (M2M) generates a Platform Specific Model (PSM) that represents the XML schema model. Finally, a model to text transformation (M2T) generates the XML document that implements the XML Schema model.

However, when we addressed the task of developing the tooling support for the proposal we faced a common problem on MDE: we need some design decisions to drive the PIM to PSM mapping. Nevertheless, according to the principles of MDE, a development process must provide for the highest degree of automation. In fact, once the PIM has been defined, the rest of the process should be completely automatic. The simplest solution in this case is to use a default value for these design decisions when coding the model transformation.

But defining a one-size-fits-all model transformation in such contexts is not enough. It may occur that some constructions are never generated on the target model. This approach could be improved by using a parameterizable transformation. Non-uniform mappings [10] and generic transformations [21] were the first works in this direction. Note that all the artefacts handled on a MDE process should be models. So, the parameters we need to drive the execution of the transformation have to take the shape of a model.

In this work we use a weaving model [1] as a container for those parameters or design decisions. Before executing the model transformation, we define a weaving model that annotates the source model. Then, both the source and the weaving model are the inputs to generate the target model. This way, different target models can be obtained from a particular source model, depending on which weaving/annotation model is used.

The results lend strong support to the idea that current MDE tools, like model transformations and weaving models are powerful enough to fulfill the requirements of XML Schema development.

The rest of this paper is organized as follows. Section 2 briefly introduces two MDE concepts: weaving models and annotation models. Section 3 presents the proposal. To that end it describes the model-driven development process for XML Schemas, the involved metamodels and the design decisions allowed when moving from the PIM to the PSM. Section 4 focuses on the implementation of the proposal by means of a case study. Section 5 summarizes related works. Finally, section 6 sums up the main conclusions as well as the future work.

## 2 Preliminaries

Before focusing on the development of the model transformation addressed in this work, we introduce some previous concepts on which our work has been based: Weaving Models and Annotation Models.

### 2.1 Weaving Models

Model transformation is essentially intended to define executable operations. Hence it is not always adapted to define and to capture various kinds of relationships between models elements. However, we often need to establish and handle these correspondences between the elements of different domains, each one defined by means of a model. The correspondences may be informal, incomplete, and preliminary. In many cases they may not be used directly to drive an executable operation. Model weaving is the process of representing, computing, and using these initial correspondences. This way, a set of correspondences between different model elements is represented as a weaving model [1].

A Weaving Model is thus a special kind of model used to establish and handle the links between models elements. This model stores the links (i.e., the relationships) between the elements of the (from now on) woven models. We illustrate this idea in Fig. 1: Mw is a weaving model that captures the relationships between Ma and Mb (the woven models), denoted by the triple [Mw, Ma, Mb]. Then, each element of Mw links a set of elements of Ma with a set of elements of Mb. For instance, the $r_2$ element of Mw defines a relationship between $a_2$ and $a_3$ from Ma, and $b_1$ from Mb.

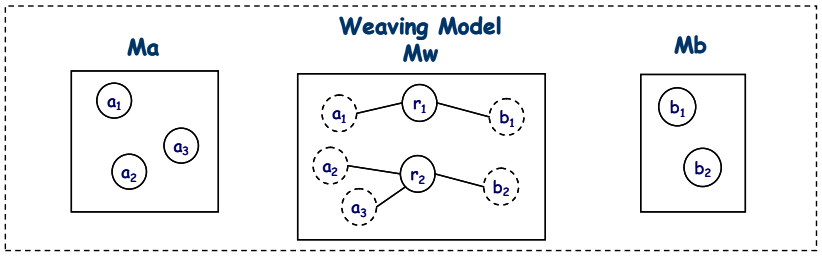


**Fig. 1.** Model Weaving overview

To create and handle the weaving models used in this work we used the ATLAS Model Weaver (AMW). The model weaver workbench provides a set of standard facilities for management of weaving models and metamodels [9]. Moreover, it supports an extension mechanism based on a Core Weaving Metamodel [8]. The Core Weaving metamodel contains a set of abstract classes to represent information about links between model elements. These classes are extended to specify new domain-specific weaving metamodels.

### 2.2 Annotation Models

MDA must support incremental and iterative development. This means that mappings between models must be repeatable. So, if a mapping requires input in addition to the source models, this information must be persistent. However, it must not be integrated into the source model, because it would mean polluting the source with information from outer domains, which is not desirable. These additional mapping inputs take the form of annotations [15].

Models are annotated or decorated to insert information that is not defined in the source metamodel. Annotation data usually is not conceptually relevant to be part of the metamodel. For example, annotations are often meta-information used for pre-processing, testing, logging, versioning, or parameterization [8].

The idea behind the use of model annotations for model transformation is the following: a model transformation specifies a set of rules that encodes the relationships between the elements from the input and output metamodels. Thus, it is defined at metamodel level, i.e., it maps elements from the input and output metamodels. It can be used to generate an output model from any model conforming to the input metamodel. That is to say that the model transformation program works for any model defined according to the input metamodel. However, in some situations this approach could be too generic and some additional considerations have to be made each time the transformation is executed. These considerations can take the form of annotations and we can collect them in an annotation model.

For instance, given a PIM and a PSM metamodel, a model transformation between them, and one terminal model conforming to the PIM metamodel, different PSM will be generated for each annotation model used to execute the transformation. This is the approach we follow in this work. Its application is showed in the following sections.

## 3 Automatic XML Schema Development in MIDAS framework

This work is framed in MIDAS [13], a model-driven methodology for IS development. Specifically, our proposal focuses on the **content** aspect of MIDAS that corresponds with the traditional concept of Database (DB). Fig. 2(a) summarizes the development process. At PIM level we use a conceptual data model represented by an UML class diagram. At PSM level, we use two different models depending on the technology selected to implement the DB: the Object Relational (OR) model and the XML model. In [4] and [5] we introduced the proposed MDE development process for XML and OR technology, respectively.

In order to support the MIDAS framework we are building a MDE environment for IS development called M2DAT (MIDAS MDA Tool). The work in this paper is integrated in the M2DAT-DB (MIDAS MDA Tool – Database) module, which provides the tooling for the content aspect of MIDAS.

All the technical solutions used to develop M2DAT share a common basis: they are part of the Eclipse Modelling Project (EMP, http://www.eclipse.org/modeling/). The EMP facilitates the deployment of any model-driven engineering process by providing a unified set of modeling frameworks, tooling, and standards

implementations. All of theses facilities are built upon a common modelling framework: the Eclipse Modelling Framework (EMF) [16]. Using EMF we have developed the model editors for each metamodel considered in MIDAS.

For depicting the class diagrams used as conceptual data model at PIM level we use UML2, the implementation of the UML 2.0 standard of EMF. To develop the PIM to PSM model transformation we use the ATLAS Transformation Language (ATL) [11]. Currently, ATL is considered the de-facto standard for M2M transformations. It offers an Integrated Development Environment (IDE) completely integrated in Eclipse. Besides, it is framed in the AMMA (ATLAS Model Management Architecture) platform that includes other facilities in the MDE context, such as the KM3 metamodeling language or the ATLAS Model Weaver (AMW) tool.

We have evaluated several proposals for code generation, such as MOFScript, JET and XPand. Finally we are using the MOFScript [17] language. It is a prototype implementation based on concepts submitted to the OMG MOF M2T transformations RFP process [18]. Since it was the first submission to the OMG RFP, it is probably the most contrasted and the most commonly used, despite the fact that recently XPand and other template-based approaches are gaining ground. Besides, the training period of MOFScript is quite short. After coding some M2M transformations, moving to M2T transformations is quite easy.

As shown in Fig. 2(a), this work focuses on the transition from the conceptual data model to the XML model. The first step towards the completion of this transition was to define the mapping rules from PIM to PSM using graph grammars. Afterwards, we coded these rules using ATL and finally, we coded the M2T transformation that returns the XML Schema. For more details see [4].

However, by the time we were coding the ATL module, we realized that some information needed to generate the target model was not included in the source model. For each execution of the transformation some *extra* information was needed. In some sense, this extra information can be shown as a way of parameterize the transformation. In a first iteration we opted for using a set of default values for these extra data. Nevertheless, it turned out that working this way, the transformation was not able to produce some constructs on the target model, whichever the source model used was. For instance, all the attributes of a particular XML element had to be grouped using the same compositor, whether it was *sequence*, *choice* or *all*. We will show a detailed example in the following sections.

The first option to overcome this drawback was to extend the source metamodel to support the modeling of this extra information. However, it is not fair to pollute the metamodel with concepts not relevant for the domain that it represents. Back to the mentioned example, the decision on how a set of PIM attributes should be mapped to an XML Schema model is a platform specific matter. It should not be considered when defining the PIM and it should not have any influence on the way we define the PIM.

Therefore, we needed a different way to collect this extra information that was related to the source model but not included in it. Since this information or *parameters* had to be available for the ATL program and considering that we were in a MDE context, the best option was to use another model (and thus to define a new metamodel): an annotation model.

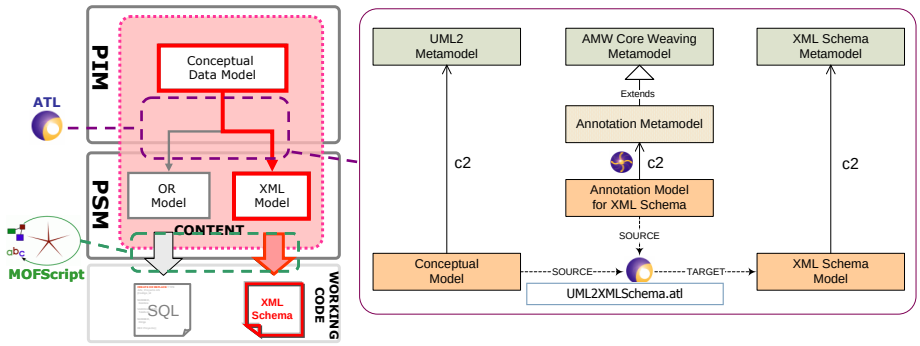


**Fig. 2.** a) Content MIDAS Dimension and b) Using *weaving* models for XML Schema

Finally, instead of defining a completely new metamodel to create our annotation models, we use a weaving model to annotate the input model. To that end, we opted for using the annotation metamodel defined as an extension to the core weaving metamodel in [8].

All this given, the resulting PIM to PSM mapping is summarized in Fig. 2(b). For every execution of the ATL transformation - in other words, for each source model (Conceptual Data Model) - we define a weaving model (Annotation Model) that contains a set of annotations. They represent the extra information needed to execute the transformation (we may refer to them as the *parameters* of the transformation). Thus, the target model is generated from the source model and the weaving model. This process allows obtaining different XML Schema models from a particular conceptual data model just by modifying the weaving model.

### 3.1 Metamodels

As Fig 2(b) shows, we use three different metamodels to map a conceptual data model to an XML Schema one: the UML2 metamodel, the XML schema metamodel and the Annotation metamodel. Since the UML2 metamodel is well known, in the following we briefly introduce the other two.

It is worth mentioning that our first step towards a model-driven approach for XML Schemas development was the definition of a UML profile for XML Schema modeling [22]. However, when we addressed the task of implementing the PIM to PSM model transformation, we decided to shift from UML profiles to Domain Specific Languages (DSL) [14]. This decision was mainly based on technical matters. As a matter of fact, technology is playing a key role in the distinction between UML based and non-UML based tools. The facilities provided in the context of the EMP and other DSL frameworks, like the *Generic Modelling Environment* (GME) or the *DSL Tools*, have shifted the focus from UML profiles to MOF-based DSLs. Therefore, regarding existing technology for (meta-) modeling and model transformations, it seemed more convenient to express the new concepts related with XML schema modeling using a new DSL. To that end we have developed a MOF-based metamodel for XML Schema modeling.

**XML Schema Metamodel.** Supporting all the constructions defined by the standard resulted in a very complex metamodel. For the sake of space, Fig. 3 shows only some parts of it. But the way they are connected helps to understand the complete metamodel that you can find at http://www.kybele.etsii.urjc.es/MtATL/.

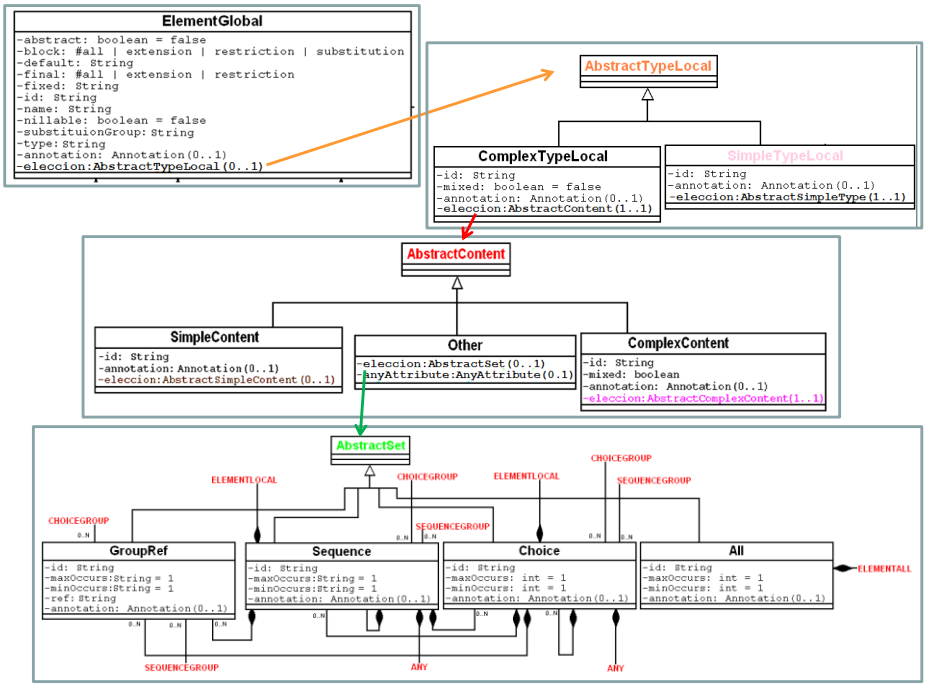


**Fig. 3.** Partial view of the XML Schema Metamodel

As Fig. 3 shows, we included a pair of modifications regarding the standard. On the one hand, we have added some hierarchies. On the other hand, some classes include an *election* property. The type of this property will be the root class of one of the added hierarchies. This way, when we set the value of the election property, we are identifying which, among the different child classes, will be the instantiated class. These modifications help on easing the management of the metamodel.

Let's show an example to better understand how these modifications work: the *election* property of the *ElementGlobal* says that its type will be an *AbstractTypeLocal* type. That is, it will be a *ComplexTypeLocal* XML element or a *SimpleTypeLocal* XML element. At the same time, the *ComplexTypeLocal* class owns an *election* property of *AbstractContent* type. This one has three children: *SimpleContent*, *ComplexContent* and *Other*. If we choose the latter, we can decide whether we will use a *GroupRef*, *Sequence*, *Choice* or *All* compositor. All together, the result is that the elements of a XML element whose type is *ComplexTypeLocal*, could be grouped using a *Sequence*, a *Choice* or an *All* compositor.

Finally, using different colours simplifies the task of identifying which hierarchy is used for defining the type of the *election* property in each specific case.

**Annotation Metamodel.** An annotation model includes a single-valued reference to the *AnnotatedModel* plus a set of annotation objects. Each annotation contains a single-valued reference to the model element plus a list of properties. The properties have an identification key and the corresponding value. The *AnnotatedModelElement* class acts as the proxy for the linked/annotated elements. That is, each record is merely a set of key-value pairs. The bottom of Fig. 4 shows the annotation metamodel used along with the core weaving metamodel [8] (top).

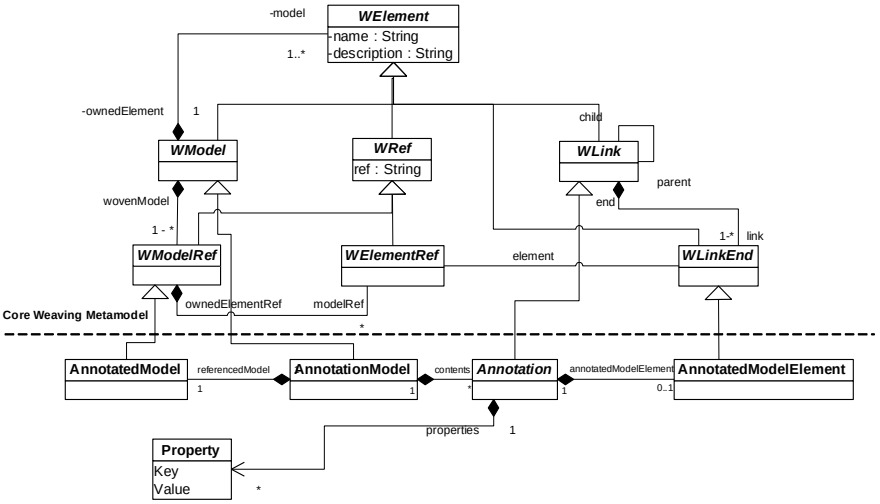


**Fig. 4.** Annotation Metamodel

### 3.2 PIM to PSM Transformation: design decisions

In the following we summarize the design decisions that can be taken to map a conceptual data model (PIM) to a XML Schema model (PSM). As we have already mentioned, in [4] we presented an initial implementation of such transformation. Here we modify some of the rules comprised in that initial version to allow the introduction of design decisions. Next, we focus only on those mapping rules. They are mainly related with the mapping of the properties of a class and the properties of a composition relationship.

- **Class Properties:** the mapping rule said that every class will be mapped to an *ElementGlobal*, which represents an element of the XML schema, plus a *ComplexTypeLocal* to define its type. The properties of that class are mapped to a sub-element (*ElementLocal*) of the *ComplexTypeLocal.* The designer can set the compositor used to group those *ElementLocals*: *all*, *choice* or *sequence* (Fig. 3). The semantics associated with each type of compositor is the following:
    - *all:* specifies that the child elements can appear in any order. Each child element can occur 0 or 1 time.
    - *choice:* allows only one of the elements contained in the declaration to be present within the containing element.

- *sequence*: specifies that the child elements must appear in a sequence. Each child element can occur from 0 to any number of times

**Default behavior:** the default compositor is *sequence*. The designer may modify this behavior by adding an annotation to the UML class. That is, by adding an annotation object in the weaving/annotation model. Such annotation will contain a property object in the form {*key = Attribute, value = Choice*} or {*key = Attribute, value = All*}.

- **Properties of a composition relationship:** composition relationships are mapped by including a sub-element within the *ComplexTypeLocal* element that maps the "WHOLE" class of the composition. This sub-element will be also a *complexTypeLocal*. It will include a set of XML sub-elements. They will map the "PART" class of the composition. The designer may choose the compositor used to group those sub-elements: *all* or *sequence*.

**Default behavior:** by default, the *sequence* compositor will be used. The designer may modify this behavior by adding an annotation to the UML association. That is, by adding an annotation object in the weaving/annotation model. Such annotation will contain a property object in the form {*key = Association, value = All*).

## 4 Case Study

In this section we use part of a case study to show the use of annotation models for model-driven development of XML Schemas. The case study is an XML DB model to store information about bibliographical references. We will start by defining the UML class diagram (section 4.1) and we will show how the annotation model (section 4.2) drives the execution of the transformation to generate the desired XML schema model.

Note that, once the conceptual data model is defined, the rest of the process is automatic. In fact, the weaving model is optional. The ATL rules have been codified to show a default behaviour if there is no annotation.

### 4.1 Conceptual Data Model

As shown in Fig 5, there are different types of bibliographical references: articles, books, chapters, translations and thesis.

Each reference has a title, a reference type, a publication date and it may has been written by more than one author and published by several publishers. In turn, a publisher may publish several references and an author may appear in more than one reference. Both, authors and publishers have a first name and a surname. The books are composed of several chapters. Each chapter belongs to one book and it may have been translated several times. Finally, each publication is composed of several articles.

The figure is a screenshot of the conceptual model represented by a class diagram using the Eclipse UML2 class diagrammer.

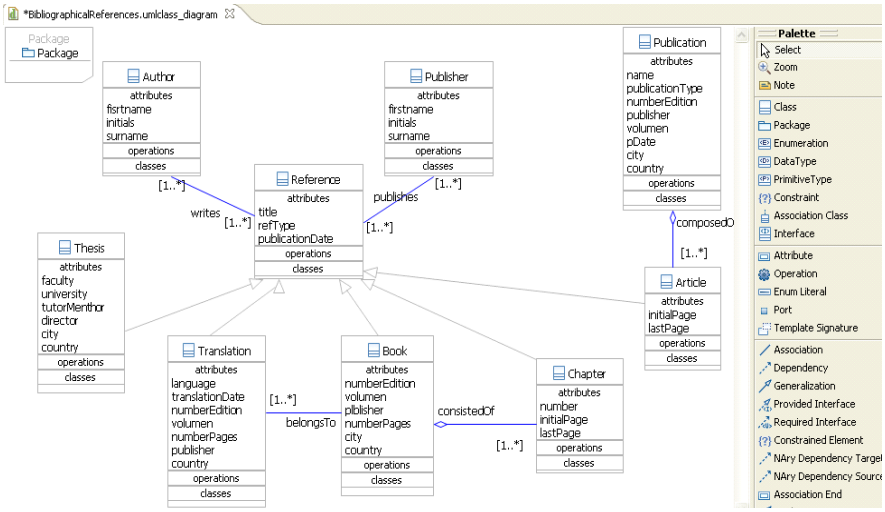


**Fig. 5.** Conceptual Data Model for the case study.

## 4.2 Annotation Model

Fig. 6 shows the weaving model used to annotate the previous class diagram. We added an annotation to the *Publisher* class. Such annotation contains a property (key = *Attribute*, value = *Choice*) that indicates that a *choice* element has to be used to map the properties of the UML class. Working this way, the designer may add an annotation to each class of the source model. The annotation sets the **compositor** (*sequence*, *choice* or *all*) used to map the properties of the class. If there is no annotation the default compositor is used (*sequence*).

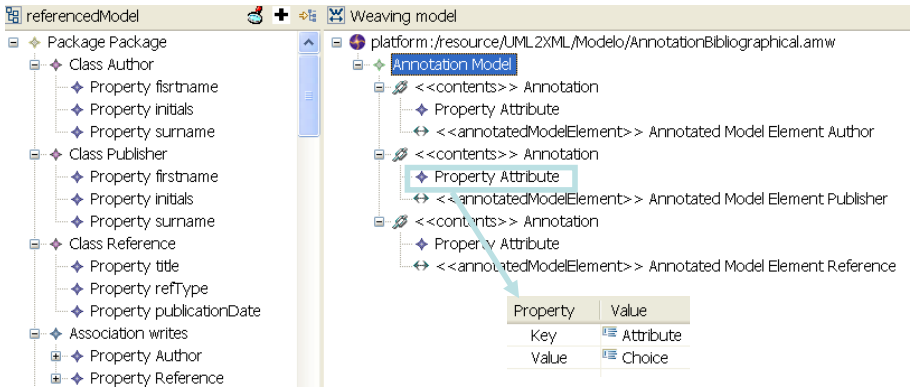


**Fig. 6.** Partial view of the weaving/annotation model for the case study.

## 4.3 Using annotations to parameterize the transformation

In this section, we show the ATL code for processing the annotations. To that end, we focus on the mapping of the *Publisher* class and its properties. This processing is

encoded in a set of rules for each type of compositor: *sequence*, *choice* and *all*, plus a set of auxiliary functions (*helpers*).

Fig. 7 shows the corresponding matching rules. For space reasons, here we show only those for using a *sequence* or a *choice* object, though the matching rule for using an *all* object is similar.

The guard of each rule checks which the decision of the designer was by calling the *mapTo()* helper.

```
rule Class2ElementGlobalSeq{
      from
            c : UML!Class ((c.mapTo() = 'Sequences')and c.GetGeneralization().oclIsUndefined())
      to
            xml : schemaXML!ElementGlobal
                  (     id <- c.name,
                        name <- c.name + '<<ElementGlobal>>',
                        Owner <- thisModule.package,
                        eleccion <- cmpTyp),
            cmpTyp : schemaXML!ComplexTypeLocal
                  (   id <- c.name + '_Type',
                        eleccion <- Other),
            Other: schemaXML!Other(
                  eleccion <- Seq),
            Seq: schemaXML!Sequences( )
}
rule Class2ElementGlobalChoice{
      from
            c : UML!Class ((c.mapTo() = 'Choice')and  c.GetGeneralization().oclIsUndefined())
      to
            xml : schemaXML!ElementGlobal
                  (     id <- c.name,
                        name <- c.name + '<<ElementGlobal>>',
                        Owner <- thisModule.package,
                        eleccion <- cmpTyp),
            cmpTyp : schemaXML!ComplexTypeLocal
                  ( id <- c.name + '_Type',
                    eleccion <- Other),
            Other: schemaXML!Other(
                  eleccion <- Seq),
            Seq: schemaXML!Choice ()
}
```

**Fig. 7.** Partial view of matching rules for mapping UML classes

As shown at the bottom of Fig. 8, the *mapTo*() helper returns the value of the designer decision by calling the *getLink*() and *getAnnotationValue*() helpers.

```
helper context UML!Class def: mapTo() : String =
      if self.getLink().oclIsUndefined() then
            'Sequences'
      else
            if self.getLink().getAnnotationValue('Attribute') = 'Sequences'
then
                  'Sequences'
            else
                  if self.getLink().getAnnotationValue('Attribute') = 'Choice'
then
                  'Choice'
                  else
                  'All'
                  endif
            endif
      endif;
```

**Fig. 8.** Helper *mapTo()*

The *getLink()* helper (Fig. 9) navigates the annotation model to return the annotation object referencing the particular property. In our case study, the annotation references the *Publisher* class. By calling the *getAnnotationValue()* helper (Fig. 10) over the

annotation object, the value of its *Attribute* property is returned. In this case, its value is *Choice*. So, the *ElementLocal* objects that will map the properties of the *Publisher* class will be grouped using a *choice* compositor.

```
helper context UML!NamedElement def: getLink() : AMW!WLink =
        AMW!WLinkEnd.allInstances()->asSequence()->select(aux | aux.element.ref = self.__xmiID__)
            ->first().refImmediateComposite();
```

**Fig. 9.** Helper *getLink()*

```
helper context AMW!WLink def: getAnnotationValue(key: String) : String =
        self.properties->asSequence()->select(prop | prop.key = key)->first().value;
```

**Fig. 10.** Helper *getAnnotationValue()*

Finally, Fig.11(a) shows the result of executing the parameterized transformation. The source models were the conceptual data model shown in Fig. 5 and the annotation model of Fig. 6.

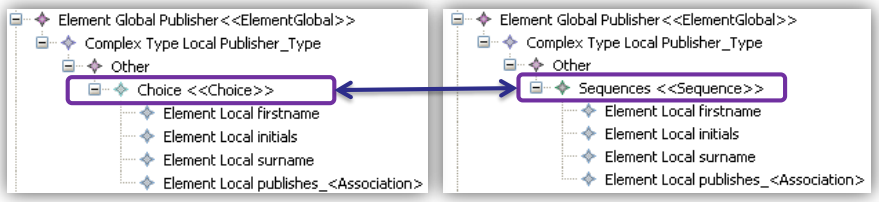


**Fig. 11.** Partial view of the XML schema model obtained: (a) using the annotation model. (b) default behavior

The code for mapping the properties of a composition relationship is very similar. As well, we encoded a set of *helpers* and matching rules for each type of compositor: *sequence* and *all*.

Fig. 12 shows the matching rule. Again, for space reasons we show just the one for using a *sequence* object.

```
rule Composite2ElementLocalSeq{
      from
            Ass: UML!Association (Ass.isAssoComposite() and not Ass.isAssoAgregation() and
            Ass.AssMapTo() = 'Sequences')
      to
            Elem: schemaXML!ElementLocal(
                  name <- Ass.name.debug('composite') + '<_is_composed_of>',
                  owner <- thisModule.resolveTemp (Ass.getPropertyAssoNoMulti().type, 'Seq'),
                  eleccion <- cmpTyp),
                  cmpTyp : schemaXML!ComplexTypeLocal
                  (eleccion <- Other),

            Other: schemaXML!Other(
                  eleccion <- Seq),

            Seq: schemaXML!Sequences(),

            Elemt: schemaXML!ElementLocal(
                  name <- Ass.getPropertyAssoLast().name,
                  owner<- Seq,
                  ref <- Ass.getPropertyAssoLast().name)
}
```

**Fig. 12.** Partial view of matching rule for mapping UML composition relationships using a sequence object.

## 5 Related works

Regarding previous works on this topic, there are two main lines to consider. On the one hand, at the end of 2000, several works focused on the use of UML to model XML Schemas. More specifically, they used UML class diagrams. Besides, they proposed to generate the XML Schema directly from the UML model [6, 7]. Working this way, the semantic gap between the abstraction levels considered is just too big. Moving from the conceptual data model to the source code is not recommendable. You will find that there are a lot of constructions that could not be obtained in the resulting code. For instance, all classes will be mapped using the same compositor. In real situations, where very complex models are used, this drawback is even more harmful. The generated XML Schema will not satisfy the needs of the designer. A language closer to the deployment platform is needed, i.e. something akin to a DSL for XML schema modelling.

A variation to this approach can be found at [19], where the mapping rules to obtain a UML model from an XML schema are defined. This proposal shows the same problem and it also lacks of any technical support.

Finally, there exists some more recent proposal focused on UML for XML Schema modelling. In [12] a comparison between them can be found. As a conclusion, we can say that none of them offer technical support.

Our proposal comprises a DSL for XML schema modelling, the mapping rules for moving from a conceptual data model to a XML Schema model, the code generation facilities to obtain the source code of the modelled Schema and the tooling to integrate these artefacts. In addition, the process can be customized by introducing some design decisions on the mapping. Moreover, in front of previous works, the one presented here is framed in a MDA framework. This fact results in additional advantages. For instance, right now we are developing the support to move from the XML technical space to the OR technical space.

## 6 Conclusion

In [4] we completed and automated our proposal for XML schema model-driven development. To that end, we defined a new metamodel for XML Schemas modelling, and we coded the M2M and M2T transformations needed.

This work has focused on the improvement one of those tasks: the transformation from conceptual data model (PIM) to XML Schema model (PSM). When validating the initial M2M implementation, we realised that we need to include certain design decisions in order to consider all the possible options when generating the XML Schema model. This article shows how we solved this problem using weaving models as annotation models. By using annotation models we can parameterize a M2M transformation without losing its generic nature. Furthermore, we are able to persist the design decisions that guided the development process through the use of models as the container for those design decisions.

The paper shows that the solution may be considered as quite simple. This is mainly due to the simplicity, the genericity and power of the AMW tool and its good coupling with the ATL model transformation solution.

The approach contributes to improve the accuracy and the quality of the models used at different stages of development as well as the subsequent code generated from them. These activities are especially important in proposals aligned with MDE because it proposes the models to be used as a mechanism to carry out the whole software development process.

At the present time we are working in two main directions. On the one hand, we are working to control entries that are mutually contradictory or inconsistent by adding OCL constraints at the metamodel.

On the other hand, we are working to support reverse engineering from the XML documents. We are defining the syntax of our XML Schema metamodel with TCS (Textual Concrete Syntax). Thus, one could not only extract an XML Schema from a model, but also inject an XML Schema to an XML Schema model.

Finally, we are working to apply the technique used here in the rest of the M2M transformations of M2DAT.

**Acknowledgments** This research has been carried out in the framework of the projects: MODEL-CAOS (TIN2008-03582/TIN), AGREEMENT-TECHNOLOGY (CSD2007-0022) both project financed by the Spanish Ministry of Education and Science and the IDONEO project (PAC08-0160-6141) financed by "Consejería de Ciencia y Tecnología de la Junta de Comunidades de Castilla-La Mancha".

## References

1. Bernstein, P A., *Applying Model Management to Classical Meta Data Problems*. In proceedings of First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA. 2003.
2. Bex, G. J., Neven, F., Van den Bussche, J., *DTDs versus XML Schema: A Practical Study.* In proccedings of Seventh International Workshop on the Web and Databases (WebDB 2004). Sihem Amer-Yahia and Luis Gravano (Eds.): pp. 79-84, Paris, France. 2004.
3. Bézivin, J., *Some Lessons Learnt in the Building of a Model Engineering Platform*. In proceedings of 4th Workshop in Software Model Engineering (WISME), Montego Bay, Jamaica, 2005.
4. Bollati, V.A, Vara, J.M., Vela, B., Marcos, E., *Una Aproximación Dirigida por Modelos para el Desarrollo de Esquemas XML*. In proceedings of XIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2008). A. Moreira, M.J. Suarez-Cabal, C. de la Riva, J. Tuya (Eds.), ISBN: 978-84-612-5820-08. Gijón, Spain. 2008.
5. Bollati, V.A., Vara, J.M., Vela, B., Marcos, E., *Una Aproximación Dirigida por Modelos para el Desarrollo de Bases de Datos Objeto-Relacionales*. In proceedings of XIV Congreso Argentino de Ciencias de la Computación (CACIC 2008). Chilecito, Argentina, 2008.
6. Carlson, D. *Modeling XML Applications with UML: Practical e-Business Applications*, Addison Wesley, Reading, ISBN 0201709155. Massachusetts, USA. April 2001.

7. Conrad, R., Scheffner, D., Freytag, J.C. *XML Conceptual Modeling Using UML*. In proceedings of International Conceptual Modelling Conference, pp. 558-571, Springer. ISBN: 978-3-540-41072-0. Salt Lake City, USA. 2000
8. Didonet Del Fabro, M., *Metadata management using model weaving and model transformation*. Ph.D. Thesis Nantes University, 2007.
9. Didonet Del Fabro, M., Bézivin, J., Valduriez P., *Weaving Models with the Eclipse AMW plugin*. Eclipse Modeling Symposium, Eclipse Summit Europe, Esslingen, Germany. 2006.
10. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A., *Transformation: The Missing Link of MDA*. In proceedings of International Conference on Graph Transformation pp: 90-105, Spinger. ISBN: 3-540-44310-X. Barcelona, Spain. 2002.
11. Jouault, F., Kurtev, I., *Transforming Models with ATL*. In proceedings of Model Transformations in Practice Workshop. (MoDELS2005) pp: 128-138. Lecture Notes in Computer Science, Vol. 3713, Briand, Lionel; Williams, Clay (Eds.), Springer. ISBN: 3-540-29010-9, Jamaica. 2005.
12. Bernauer, M., Kappel,G., Kramler, G. *Representing XML Schema in UML - A Comparison of Approaches*. In proceedings of 4th International Conference on Web Engineering (ICWE2004), p. 440 – 444. LNCS 3140, Springer, ISBN: 978-3-540-22511-9. Munich, Germany. 2004
13. Marcos, E. Vela, B., Cáceres, P., Cavero, J.M., *MIDAS/DB: a Methodological Framework for Web Database Design*. In proceedings of Conceptual Modeling for New Information Systems Technologies, DASWIS conference, pp. 227-238. LNCS 2465, Springer-Verlag, ISBN: 978-3-540-44122-9. Yokohama, Japan. November, 2001
14. Marjan, M., Jan, H., Anthony, M.S., *When and how to develop domain-specific languages*. ACM Comput. Survevs. Vol. 37, Issue 4. pp. 316-344, ISSN: 0360-0300. 2005.
15. Mellor, S., Scott, K., Uhl, A., Weise, D.: *Model-Driven Architecture.* In proceedings of Advances in Object-Oriented Information Systems. pp. 233-239 (2002).
16. Moore, B. Dean, D. Gerber, A. Wagenknecht, G., Vanderheyden, P. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. Retrieved from: http://ibm.com/redbooks. February 2004.
17. Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., Berre, A.-J., *Toward Standardised Model to Text Transformations.* Model-driven Architecture – Foundations and Applications, pp. 239-253, LNCS. Springer Berlin / Heidelberg, ISBN: 978-3-540-30026-7. 2005.
18. Object Management Group (OMG). (2004c). *MOF Model to Text Transformation Language RFP*. Retrieved from http://www.omg.org/cgi-bin/doc?ad/04-04-07.
19. Salim, F.D.; Price, R.; Krishnaswamy, S.; Indrawan, M. *UML documentation support for XML schema*. In proceedings of Australian Software Engineering Conference, pp: 211-220. 2004.
20. Selic, B., *The pragmatics of Model-Driven development*. IEEE Software, Vol. 20, 5, September.-October. 2003.
21. Varró, D. y Pataricza, A.*, Generic and Meta-Transformations for Model Transformation Engineering*. UML 2004. In proceedings of 7th International Conference, pp: 290-304. LCNS, Vol. 3273, Springer, ISBN: 978-3-540-23307-7. 2004.
22. Vela, B., Acuña, C., Marcos, E., *A Model Driven Approach for XML Database Development*. In proceedings of 23rd. International Conference on Conceptual Modelling (ER2004), pp. 780-794 LNCS 3288. Springer Verlag, ISBN: 978-3-540-23723-5. Shanghai, China. November, 2004.
23. W3C, *XML Schema Working Group. XML Schema Parts 0-2Primer, Structures, Datatypes]*. W3C Recommendation. Retrieved from: http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/ y http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/, 2004.