Configuring ATL Transformations in MOSKitt[1]

Javier Muñoz[1], Miguel Llacer[1], Begoña Bonet[2]

[1] Prodevelop S.L,
Plaza Don Juan de Villarrasa, 14 – 5
46001 VALENCIA (SPAIN)
{jmunoz, mllacer}@prodevelop.es
[2] Conselleria de Infraestructuras y Transporte, Generalitat Valenciana,
Av. Blasco Ibáñez, 50
46010 València
bonet_beg@gva.es

Abstract. Model transformations need to be configured in order to satisfy particular user requirements in real scenarios. This paper introduces an strategy for configuring ATL transformations. This strategy, that is currently in practice in the MOSKitt tool, follows a model driven approach, where both the specification of configuration options and the user selections of those options are implemented as models. Additionally, the user interfaces that are provided by MOSKitt for selecting the configuration options are briefly introduced.

Keywords: ATL, model transformation, MOSKitt, model driven engineering

1 Introduction

Model transformations play a key role in model driven software development approaches. Different kinds of model transformation (refinements, refactoring, translations, etc.) are applied during the development process in order to achieve the desired products (source code, reports, etc.). When developing a model transformation, the transformation analyst may identify different options for transforming some elements. One of these options may fit better in some scenario or satisfy some requirement whether other option may be more suitable in other scenarios. Therefore, in an industrial context, mechanisms for configuring the transformation in order to select between different options must be provided by model driven software engineering tools.

In this paper we introduce an approach for supporting the configuration of ATL [1] transformations. The approach is based on the definition of a configuration model that is used as input by the transformation in order to generate the output following the

1 The budget provided by Generalitat for funding these actions may have fundings of the European Regional Development Fund (ERDF) through the Programa Operativo de la Comunitat Valenciana 2007-2013

user guidelines. This strategy has been defined in the context of the MOSKitt project [2] and it is currently implemented in the Eclipse-based MOSKitt[2] tool.

The MOSKitt project aims to provide a set of open source tools and a technological platform for supporting the execution of software development methods which are based in model driven approaches, including graphical modeling tools, model transformations, code generation and collaboration support. This project is led by the Conselleria de Infraestructuras y Transporte (CIT) (Ministry of Infrastructures and Transport) of the Regional Government of Valencia, in Spain. The project was initiated in 2007 and their results are currently in practice by CIT and other companies and ministries in Spain.

The paper is structured as follows. Next in this section a running case study is introduced to illustrate the contents proposed approach. In Section 2 the strategy for implementing the configuration of ATL model transformation is introduced. Section 3 discusses how to deal with the configuration information in ATL transformation. Then, Section 4 presents the user interfaces that are currently available in MOSKitt to select the configuration options. Finally, Section 5 includes some conclusions and outlines our future works in this area.

1.1 Case Study

In order to describe the approach for configuring model transformation, the classical transformation from UML2 to database models will be used. In this transformation we can find some patterns in the input models where different options can be selected. In this paper we will use as an example the transformation of UML2 Generalization elements into database elements.

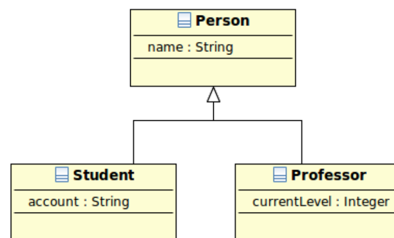

Fig. 1. A UML2 hierarchy of classes with a general Class and two specific ones.

Figure 1 shows a simple UML2 hierarchy where the general element *Person* is specialized into two more specific entities *Student* and *Professor*. In this scenario we can identify at least three options for transforming this model:

1. **OnlyParentTable**: Transforming all the hierarchy into one Table (*Person*) with a Column for each property of the UML2 classes (*name*, *account*,

2 http://www.moskitt.org/eng

currentLevel), adding an extra Column for specifying the kind of *Person* (Person, Student or Professor).

2. **OnlyChildTable**: Transforming only leaf Classes into Table elements (*Student* and *Professor* in our example) and converting properties into columns (*Professor* table will have *name* and *account* columns whereas *Student* table will have *name* and *currentLevel* columns).
3. **AllTables**: Transforming every Class into a Table with a Column for every property and adding foreign keys from the Tables derived from specific Classes to the Tables derived from the general Classes (from *Student* to *Person* and from *Professor* to *Person*).

Of course, other UML2 elements can have transformation configuration options like, for instance, many to many associations, compositions and enumerations.

2 Approach for Configuring Model Transformations

The approach for configuring model transformations introduced in this paper, which is depicted in Figure 2, follows a model driven strategy. Models are used both for capturing the information about which are the transformation options and for specifying the options that a user has selected for executing the transformation. Attending the classification of configuration techniques that is introduced in [3], we follow a Domain Specific Language approach.

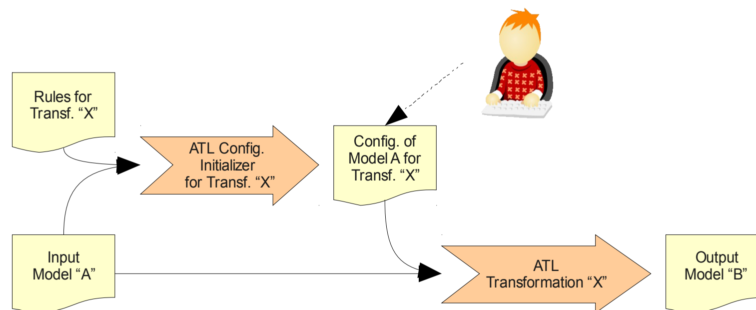

Fig. 2. Schema of the approach for configuring model transformations with ATL.

The options for configuring each configurable kind of element (like Generalization) are specified in a Rules Catalog model. This model contains a list of configuration patterns. A configuration pattern defines

- which are the configured elements (e.g. Generalization).
- which are the configuration rules (e.g. OnlyParentTable, OnlyChildTable, AllTables).
- which is the default configuration rule (e.g. AllTables).
- which are the configuration rule parameters (there are not in our example).

The metamodel of the Rules Catalog model is depicted in Figure 3. A Catalog is composed of ConfigurationPatterns. Different transformation Rules can be applied to a ConfigurationPattern.

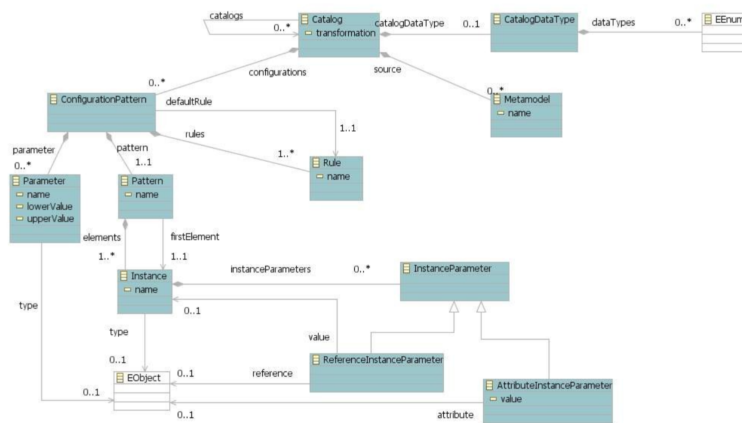

Fig. 3. Metamodel for describing the different rules available in a model transformation.

The selected options for an specific execution of the transformation are specified in a transformation configuration model. In our approach, ATL transformations automatically generates the initial configuration with the default options selected for the input model. This configuration model specifies for each configurable element in an input model which rule is selected. Users can modify the transformation configuration model to select the configuration rules that satisfy their needs. The metamodel of the Configuration model is depicted in Figure 4.

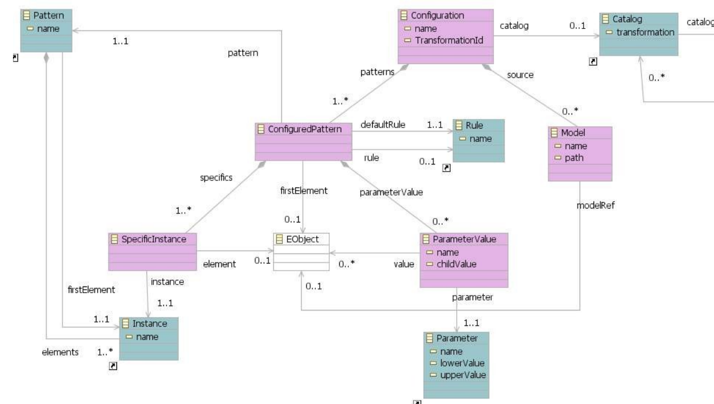

Fig. 4. Metamodel for specifiying the configuration of a transformation for an input model

Finally, the ATL transformation reads the configuration as an input model and modifies their behavior according to the user selections. It is important to note that it

is the transformation responsibility taking into account the options that are selected in the configuration model in order to produce the expected output.

4 Configuring ATL Transformations

As introduced in the section above, two ATL transformations must be implemented to support the configuration of model transformations: the transformation for generating the initial configuration options and the transformation which must support the configuration.

The first transformation (catalog2configuration.atl) takes as input a model with the catalog of configuration rules and the model to be configured and produces the initial configuration:

```
create OUT : CONFIGURATION from IN : CATALOG, INMODEL : UML;
```

First of all, the transformation selects those elements in the actual UML model that can be configured by the *GeneralizationPattern*. Those elements are the sent to the rule *TransformPatternElements(confPattern, elem)*, which generates the default configuration.

```
rule GeneralizationPattern2ConfiguredPattern {
     from
        i: CATALOG!ConfigurationPattern (i.isGeneralizationPattern())
     do {
        for (elem in UML!Class.allInstances()->select(c |
              c.oclIsTypeOf(UML!Class) and not c.isAbstract and
              ((UML!Generalization.allInstances()->exists(g | g.general = c)) or
              c.powertypeExtent.notEmpty()))) {
              thisModule.TransformPatternElements(i, elem);
        }
     }
}
```

The *TransformPatternElements(confPattern, elem)* rule generates a ConfiguredPattern element, which holds the selected configuration option, and an SpecificInstance element, which references in this case the UML element that is being configured. As it is shown in the code of the rule, elements for holding the parameter values are also generated if necessary (not in our example).

```
rule TransformPatternElements (confPattern : CATALOG!ConfigurationPattern, elem :
UML!Element) {
     to
        o     :           CONFIGURATION!ConfiguredPattern,
        o1    :           CONFIGURATION!SpecificInstance
     do {
        o.pattern <- confPattern.pattern;
        o.defaultRule <- confPattern.defaultRule;
        o.firstElement <- elem;
        o.specifics <- o.specifics.append(o1);
```

```
        o1.element <- elem;

        for (inst in confPattern.pattern.elements) {
              o1.instance <- inst;
        }
        for (param in confPattern.parameter->select(e |
                          e.lowerValue = e.upperValue and e.upperValue = 1)) {
              if (param.name = 'Unique Constraint') {
                          if (elem.oclIsTypeOf(UML!Association)) {
                            if (elem.isOneToOne() and elem.betweenClasses()) {
                              thisModule.generateParameterValue(o, param);
                            }
                          }
              }
              else {
                          thisModule.generateParameterValue(o, param);
              }
        }
        thisModule.resolveTemp(confPattern.eContainer(),'o').patterns <-
        thisModule.resolveTemp(confPattern.eContainer(),'o').patterns.append(o);
    }
}
```

As it has been introduced in the previous section, is a transformation responsibility to deal with the configuration information to generate the suitable output. A simple but effective strategy to provide the desired behavior is to implement a different ATL rule for each configuration option. Using this approach, additional conditions are added in the FROM part of the rule checking which configuration option must be applied.

```
rule GeneralizationParent2Table {
       from
           i : UML!Class (
                i.oclIsTypeOf(UML!Class) and
                i.executeGeneralizationRule('OnlyParentTable') and
                …  )
       do {  …  }
}
```

This strategy performs well when only one configuration pattern is applied to an element. In case that several kind of options can be selected for one element, we should implement as rules as possible combination of options. For instance, if Generalization could also be configured using another criteria with two options, we should implement (3*2) six rules. In this scenario, a potential strategy to avoid the explosion in the number of rules would be to implement a unique rule with a DO section in charge of discriminating the configuration options and calling rules that

implement the different transformation actions. This behavior may be implemented more naturally adding a transformation phasing [4] mechanism to ATL.

Our configurable ATL transformation generates DB models and traces taking as input UML2 models and the configuration model. This transformation can be found in the public MOSKitt repository[3].

```
create OUT : DB, trace : Trace from IN : UML, modelConf : CONF;
```

In the configuration of the generalization, we follow the approach where a rule is implemented for every transformation configuration option. Therefore, we have implemented three different rules:

```
rule GeneralizationParent2Table {
     from
        i : UML!Class (
              i.oclIsTypeOf(UML!Class) and
              i.isHierarchyParent() and
              i.isPersistentClass() and
              i.executeGeneralizationRule('OnlyParentTable')
        )
     do {

        -- generate Parent Table
        thisModule.generateTable(i);
         --generate additional attribute: 'tipo_<class_name>'
         thisModule.createAdditionalColumn(i, i.getTable());
        -- process child class
        if (i.getTable() <> OclUndefined) {
           thisModule.processChildFromParentTable(i, i.getTable(), Sequence{});
        }
     }
}

rule GeneralizationChild2Table {
     from
        i : UML!Class (
              i.oclIsTypeOf(UML!Class) and
              i.isHierarchyParent() and
              i.isPersistentClass() and
              i.executeGeneralizationRule('OnlyChildTable')
        )
     do {
     thisModule.processChildFromTakeDownProperties(i, Sequence{i}, Sequence{});
     }
}

rule Generalization2Tables {
     from
```

3 http://subversion.moskitt.org/gvcase-uml2db/trunk/es.cv.gvcase.linkers.uml2db.transf/src/es/cv/gvcase/linker/uml2db/transf/launcher/Transformations/uml2db.atl

```
        i : UML!Class (
              i.oclIsTypeOf(UML!Class) and
              i.isHierarchyParent() and
              i.isPersistentClass() and
              i.executeGeneralizationRule('AllTables')
        )
    do {
        -- generate table
        thisModule.generateTable(i);
        -- process child class
        thisModule.processChildFromAllTables(i);
    }
}
```

The creation of the tables (and columns, if necessary) has been implemented in ATL called rules in order to facilitate the maintenance of the code

5 User Interface for Configuring Model Transformations

MOSKitt provides a forms editor for editing the transformation configuration models. This editor, which is depicted in Figure 5, shows in the left side a tree with the model elements that can be configured. When a configuration is selected in the editor, users can edit their properties in the right side of the editor, including the desired configuration rule and the configuration parameters, if any is available. Once the configuration model has been edited, users can initiate the execution of the transformation from this editor.

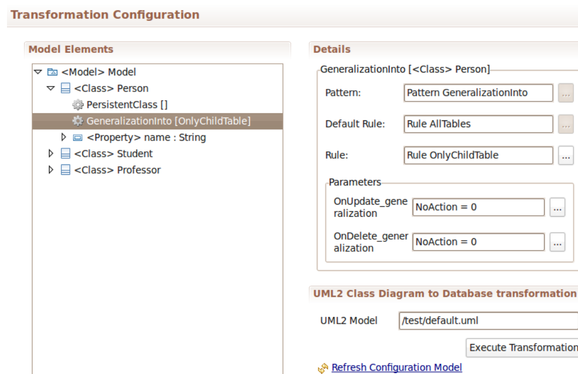

Fig. 5. Eclipse form editor for selecting the transformation configuration rules to be applied.

Additionally, the functionality of the transformations configurations editor has been integrated as a property sheet of the graphical editors. This property sheet shows the configuration options for the element that is selected in the graphical editor.

Therefore, users can select the transformation configuration options while they are editing the model.

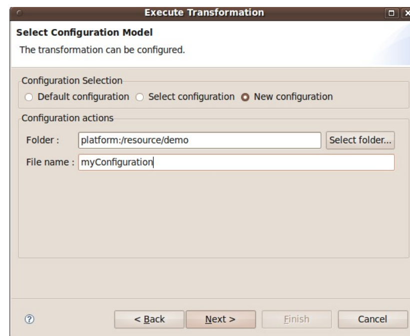

Fig. 6. Wizard page for selecting the transformation configuration options.

The configuration of ATL transformations have also been integrated in the MOSKitt wizard for executing transformations. This wizard provides a sequence of pages (1) for gathering transformation parameters, (2) for providing output about the validation of inputs, (3) for configuring the transformation and (4) for showing the results of the transformation configuration. In the page for configuring the transformation, that is shown in Figure 6, users can select a previously used configuration model, can create a new configuration model or can use the default configuration selections. If a new configuration wants to be created, the configuration editor with the default configuration model is automatically opened.

6 Conclusions and Future Works

The approach for configuration ATL model transformations that is introduced in this paper is currently applied in the open source MOSKitt tool. Several ATL transformations can be configured to modify their behavior attending user requirements, including the UML2 to DB transformation with 19 configuration options. This approach is also applied to other kind of transformation like, for instance, code generation transformations implemented with XPand.

As future works, the usability of the editor of configuration models will be analyzed and increased as much as possible, since it is a key piece from the users point of view. Additionally, we plan to extend the rules catalog metamodel to be capable of expressing dependencies (requirements and exclusions) between configuration rules.

References

1. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of Lecture Notes in Computer Science., Springer-Verlag (2006) 128–138

2. García Hernández, M. Proyecto gvCASE: Desarrollo de una herramienta CASE para dar soporte a gvMétrica. Jornadas sobre Tecnologías de la Información para la Modernización de las Administraciones Públicas (TECNIMAP 2007). Gijón, Spain. 2007

3. D. Wagelaar and R. van der Straeten. A Comparison of Configuration Techniques for Model Transformations. In 2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13, pages 331–345, 2006. 17

4. Cuadrado, J. S. and Molina, J. G. 2007. A phasing mechanism for model transformation languages. In *Proceedings of the 2007 ACM Symposium on Applied Computing* (Seoul, Korea, March 11 - 15, 2007). SAC '07. ACM, New York, NY, 1020-1024. DOI= http://doi.acm.org/10.1145/1244002.1244223