SADIO Electronic Journal of Informatics and Operations Research

http://www.dc.uba.ar/sadio/ejs

vol. 9, no. 1, pp. 49-66 (2010)

# Model transformation as a mechanism for the implementation of domain specific transformation languages

Jerónimo Irazábal[1,2] Claudia Pons[1,2,3] Carlos Neil[3]

[1] LIFIA
Facultad de Informática, Universidad Nacional de La Plata
Calle 50 esq.120. La Plata. Buenos Aires. Argentina
Emails: [jirazabal,cpons]@lifia.info.unlp.edu.ar

[2] CONICET, Consejo Nacional de Investigaciones Científicas y Técnicas.
Buenos Aires, Argentina.

[3] Universidad Abierta Interamericana (UAI)
Av. Montes de Oca 745 - (C1270AAH) Ciudad de Buenos Aires. Argentina
Email: carlos.neil@vaneduc.edu.ar

**Abstract**

Model Driven Engineering proposes a software development process in which the key notions are models and model transformations. There are already several proposals for model transformation specification, implementation, and execution. In this paper we introduce the notion of domain specific transformation language (DSTL). A DSTL is a transformation language tailored for a specific domain; in contrast to well known transformation languages, such as QVT or ATL, the DSTL's syntax and semantics are directly related to a specific domain and/or kind of transformation. A DSTL makes transformations easer to write and understand, the code is intuitive and the users do not need to know a generic transformation language. Also we analyze a novel way to define its semantics. Our proposal consists in using transformation languages themselves to the implementation of such domain specific languages. We illustrate the proposal through an example in the database domain.

**Keywords**: model driven engineering, model transformation language, domain specific language, semantics, ATL

## 1 Introduction

Modeling is significant for dealing with the complexity of computer systems during their development and maintenance processes. Models allow engineers to precisely capture relevant aspects of a system from a given

perspective and at an appropriate level of abstraction. Then, model transformations provide a chain that enables the automated development of a system from its corresponding models. Model Driven Engineering (MDE) [Kleppe et al., 2003; Stahl and Völter, 2006; Pons et al., 2010] proposes a software development process in which the key notions are models and model transformations.

Models can be expressed using different languages. Unlike general-purpose modeling languages (GPMLs), such us the UML, Domain-specific modeling languages (DSMLs), such as the RDBMS language, can simplify the development of complex software systems by providing domain-specific abstractions for modeling the system in a precise but simple and concise way. DSMLs have a simpler syntax (i.e., few constructs focused to the particular domain) but its semantics is much more complex (because all the semantics of the particular domain is embedded into the language).

In this process, software is built by constructing one or more models, and successively transforming these into other models, until finally the output consists of program code that can be executed. A model transformation is a set of transformation rules that together describe how a model written in the source language is mapped to a model written in the target language. Model transformations are specified using a model transformation language. There are already several proposals for model transformation specification, implementation, and execution, which are beginning to be used by Model-Driven Engineering practitioners [Czarnecki and Helsen, 2006]. The term "model transformation language" comprises all sorts of artificial languages used in model transformation development including general-purpose programming languages, domain-specific languages (DSLs) [Mernik et al., 2005], modeling and meta-modeling languages and ontologies. Examples include languages such as the standard QVT (Query/View/Transformation) [OMG/QVT, 2005], ATL (ATLAS Transformation Language) [ATLAS team, 2006; Jouault and Kurtev, 2006] and RubyTL [Sánchez Cuadrado et al., 2006].

These languages are specific for defining model transformations but they are independent of any modeling domain; so they contain complex constructs referring to pattern matching mechanisms, control structures, etc. This can eventually compromise the primary aims against which the DSML was built: domain focus and conciseness. Therefore, an extra level of specialization can be realized on them. This means, we can define a transformation language specifically addressed to a given transformation domain, i.e., a Domain Specific Transformation Language (DSTL). For example, we can create a language focused on either the definition of transformations between database models or the definition of transformations between business models, among others.

In this context if we would like to take advantage of a very specific transformation language we face the problem of implementing such a new language. There exists powerful frameworks for the definition of domain specific languages, such as Eclipse [ISIS-GME, 2008; Gronback, 2009], Microsoft DSL Tools [Greenfield et al., 2004; Cook et al., 2007] and AMMA [Bézivin et al., 2006]. These frameworks are mainly focused on the definition of the syntax (both abstract and concrete) of the DSL, while less attention is devoted to the semantics of the language. In general the semantics is indirectly defined by the code generation mechanisms that allow us to specify which the code associated to each modeling artifact is. Nevertheless, the AMMA framework is an exception, since it takes advantage of the MDE ideas. Within the AMMA framework the semantics of a DSL can be defined in a more abstract manner either in terms of Abstract State Machines (ASMs) or based in another language. In [Jouault et al., 2006], is described the application of the AMMA framework to the implementation of the languages SPL and CPL for the telephony domain.

In the present work we introduce the proposal of defining domain specific transformation languages (DSTLs) and also we analyze a novel way to define their semantics. Our proposal consists in using transformation languages themselves to the implementation of such DSTLs.

This paper is organized as follows. Section 2 presents the main features of the proposal to define domain specific languages using transformation languages. Section 3 illustrates the use of the approach by the definition of a DSTL for the transformation of extended relational models. Section 4 shows relevant parts of the ATL-based implementation of such DSTL. Section 5 discusses an alternative implementation approach. Section 6 compares this approach with related research and finally Section 7 presents the conclusions.

## 2 DSL implementation schema

The AMMA framework [Bézivin et al., 2006] allows us to define the concrete syntax, abstract syntax, and semantics of DSLs. In [Jouault et al., 2006; Barbero et al., 2007; Di Ruscio et al., 2009] the reader can analyze a number of scenarios where the AMMA framework has been used to define the semantics of DSLs in terms of other languages or in terms of Abstract State Machines (ASMs).

Our proposal has similar goals to the AMMA framework, but we present a novel alternative, where the language semantics is achieved by means of a transformation written in the ATL language. Our schema can be seen as the interpretation of the DSL into the ATL transformation language. Our implementation approach consists in the generation of a transformation *T* (written in ATL) that takes two inputs: an instance of the DSL metamodel (*T1*), that is, a domain specific transformation written in the domain specific language (such as a transformation between databases) and a model (*M1*) belonging to the specific domain (e.g., a concrete user database model). The output of such transformation *T* is the model that is expected to be produced by the application of the domain specific transformation on the input model (*M2*). Figure 1 shows the transformation scenario.

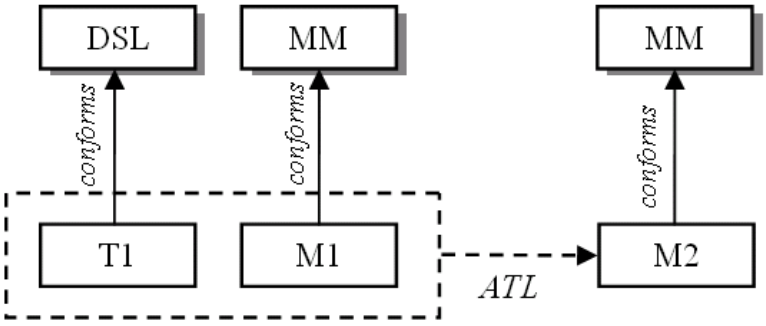



**Figure 1.** Transformation scenario.

In our implementation we directly deal with the abstract syntax of the DSL, which implies an important simplification. Nevertheless, this simplification can be easily relaxed in order to also consider concrete syntaxes: for example by using the TCS (Textual Concrete Syntax) language, which is provided by the AMMA framework to this particular purpose.

## 3 A domain specific transformation language for transforming relational models

In this section we first present the simplified version of the relational model that we will use; then we define a language that allows us to transform relational database models in a wide spectrum. Such language deals with the data model, as well as with the scripts and the existing data that populate the base. Finally, we illustrate the effectiveness of the language through its application to the transformation of a simple database model.

Notice that we do not intent to make a contribution to the field of database transformations. There are many approaches for refactoring already defined – see for example: http://databaserefactoring.com/. We have selected this domain due to its simplicity to show the applicability and advantages of DSTLs.

## 3.1 The relational model

Due to the fact that the transformation language is expected to express the transformation of the whole spectrum of a database model (i.e., the data model, the scripts and the concrete user's data), the source language of the transformation should be able to represent all those elements. Consequently the metamodel that we define in this work is richer than the classical relational metamodel described in [OMG/QVT, 2005], which is particularly restricted to the M1 level of the OMG's 4-levels metamodeling architecture [OMG/MOF, 2003]. Therefore, our metamodel contains additional meta-classes to represent both scripts and data values as well. Figure 2 shows the modified relational metamodel.

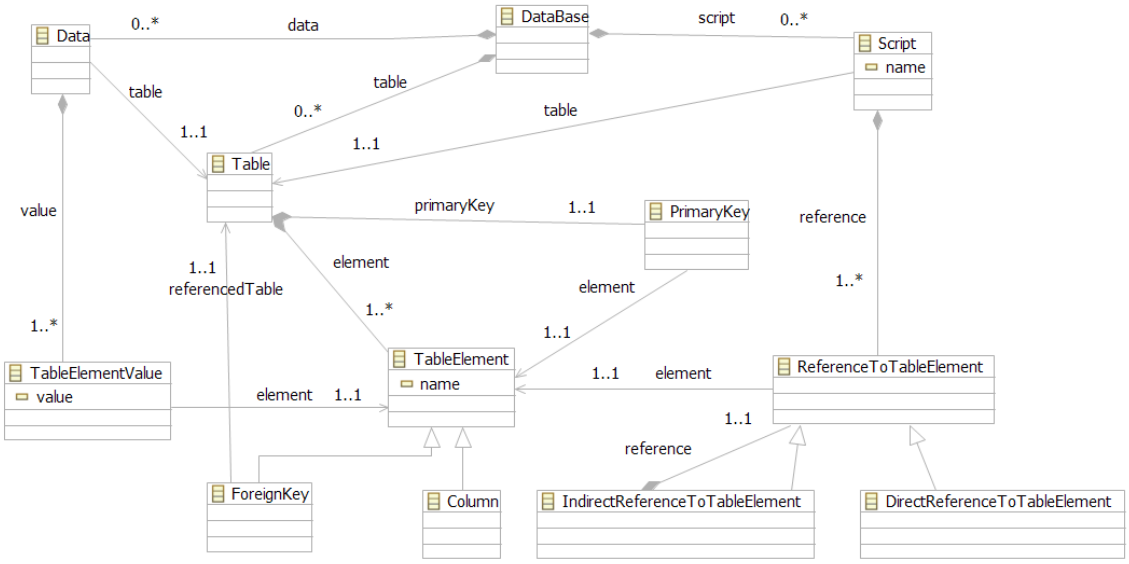



**Figure 2.** Simplified relational metamodel including scripts and data values

For the sake of clarity, a number of simplifications have been applied to this meta-model; the most relevant ones are: unique data type (string of chars), simple key and single script semantics per interpretation. All these simplifications can be removed without major changes in the proposal.

## 3.2 A DSTL fitting the relational model

We define a simple domain specific transformation language (DSTL), with the aim of transforming relational databases. This language will express the transformation of the three elements we mentioned before: the data model, the scripts and the data values. This specific language allows us to denote the most usual kinds of transformations in the databases domain. As an example we include here the description of only three transformations: *changeName*, *extractCommonData* and *factorize*. The abstract syntax of the DSTL is as follows:

```
<transformation> ::=
      changeName <table> <string>                    |
      extractCommonData <table> <element> <table>    |
```

```
    factorize <table> <element> <table> <element>* |
    <transformation>;<transformation>

<table> ::=     table <string>
<element> ::= column <string> | foreignKey <string>
<string>  ::=   a | b | c | … | <string> <string>
```

Due to the fact that we will use model transformations to implement this DSTL, we need to have the DSTL's abstract syntax defined by a metamodel. Figure 3 displays the metamodel of our relational DSTL.

After defining the syntax of our language we need to define its semantics. As an initial step, we describe the semantics using just natural language by means of definitions that transmit an intuitive understanding of the meaning of each syntactic construct. However, much formality is required in order to guarantee the correct implementation of the DSTL. Such formal definition of the semantics will be addressed in the following sections.

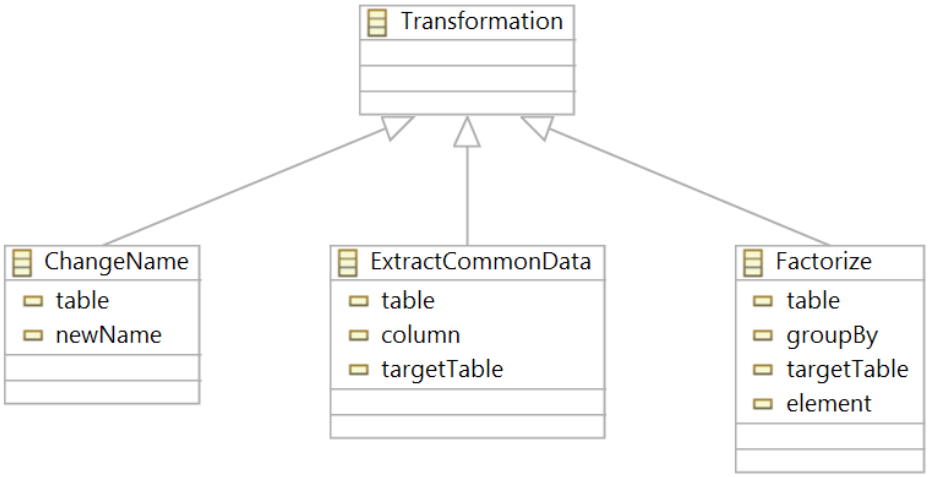



**Figure 3.** Metamodel of the domain specific transformation language

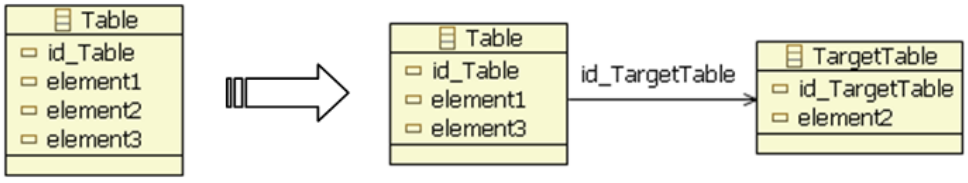
- ***changeName***:

  This is a very simple transformation, in which its effect consists of changing the name of the input table.

Next transformations are considerably more complex and they will receive a more exhaustive treatment:

- ***extractCommonData*:**

  This transformation specifies the splitting of a table into two tables with the goal of avoiding data duplication. The source of this transformation is a table and a selected column (containing duplicated data). The transformation creates a fresh table. Existing data is collected from the input table and then it is stored in the fresh table in a grouped way (avoiding the duplication of data). In parallel the references contained into the scripts are consistently modified so that the behavior of the scripts keeps unaltered. Figure 4 illustrates the effect of this transformation at model level.




```
<ExtractCommonData table="Table" column="element2" targetTable="TargetTable"/>
```

**Figure 4**. Effect of the *extractCommonData* transformation.

In order to make the behavior of this transformation more comprehensible, we describe it from an operational point of view: any algorithm performing this transformation should carry out, in some concrete way, the following steps.

1) To create the target table (in the case the table does not exist);
2) To replace the selected column in the source table by a foreign key to the target table;
3) To replace the direct references to the selected column by an indirect reference to the column in the target table;
4) To move the data from the column of the source table to the target table, avoiding data duplication;
5) To modify the data stored in the source table, establishing the value of the added foreign key (step 2) as the value of the primary key of the target table, corresponding to the value of each moved data (step 4).

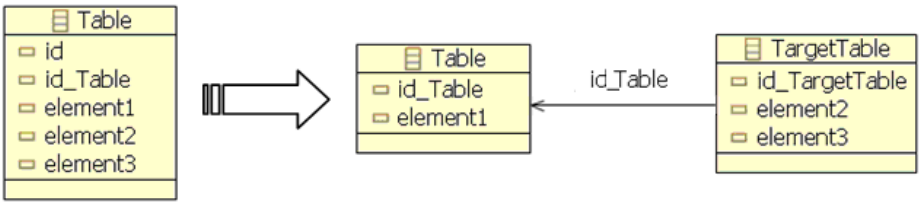
- **Factorize:**

In a similar way to the previous transformation, the factorize transformation states the splitting of a table into two tables with the goal of avoiding data duplication. The main difference with respect to the *extractCommonData* transformation consists in that this last transformation generates a target table with references to the source table. Direct references to removed elements of the source table will be transformed to direct reference to the corresponding element in the target table. The data from the source table will are transformed in order to keep only one value for each different value in the grouping column. Such column will become the new primary key of the source table (previous primary key is removed).

As expected, the evaluation of any transformed script on the target database will present no observable difference with respect to the evaluation of the corresponding source script on the source database. The effect of the transformation on the data model is illustrated in Figure 5. In terms of an algorithm, we have the following steps:

1) To create the target table (in the case the table does not exist);
2) To remove the elements in the source table;
3) To remove the primary key from the source table and to set up the grouping column as the new primary key;
4) To replace direct references to removed elements with a direct reference to the corresponding element in the target table;
5) To keep only one value for each different value of the new primary key (duplicated data is removed).

6) To move the existing data from the source table to the new table, replacing the value of the external references to the source table by the value of the grouping column in the source table.




```
<Factorize table="Table" groupBy="id_Table" TargetTable="TargetTable">
   <element>element1</element>
   <element>element2</element>
</Factorize>
```

**Figure 5.** Effect of the *factorize* transformation.

As it is expected, the evaluation of any transformed script on the transformed database will present no observable difference with respect to the evaluation of the corresponding source script on the source database.

## 3.3 Example

In this section we show the applicability of the domain specific transformation language. To this purpose we present a very simple example consisting of a minimal database containing a single table named “Book”. This table has seven columns: ISBN, title, editorial, comments, availability, chapterTitle and chapterPages.

By using our DSTL we will transform this database to a behavioral equivalent database without data duplication. In order to specify such transformation we make use of a concrete syntax based on XML and directly supported by the AMMA framework, as follows:

```
<ExtractCommonData table="Book" column="editorial" TargetTable="Editorial"/>
<Factorize table="Book" groupBy="isbn" TargetTable="Chapter">
   <element>chapterTitle</element>
   <element>chapterPages</element>
</Factorize>
```

Figure 6 displays the source model (to the left hand side) and the target model – i.e., the result of the transformation (to the right hand side).

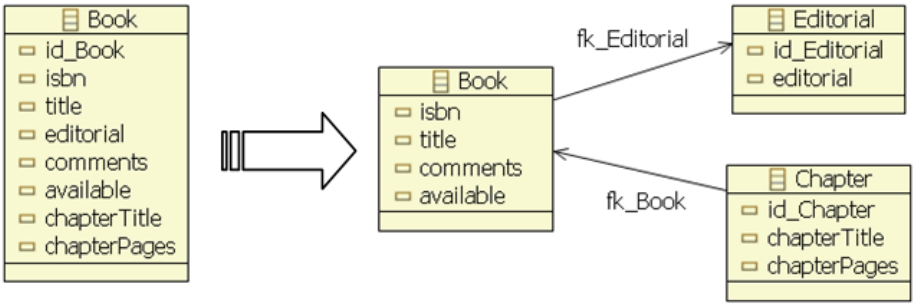



**Figure 6.** The data model before and after the transformation application.

After applying the first transformation, the editorial information is not longer a column in the "Book" table. The editorial information becomes a new entity in the target database, i.e. the "Editorial" table. The second transformation prevents us from having the general information of the book duplicated for each chapter. After performing the transformation, the book general information becomes separated from the chapters by means of the new table entity "Chapter".

## 4 DSTL Implementation

In this section we present the implementation of our DSTL by using the model transformation language ATL. The implementation consists of a transformation, written in ATL that takes two inputs: a relational database (conforming the relational metamodel in Figure 2) and a transformation specified in the relational transformation language (conforming the DSTL metamodel in Figure 3). The output of such transformation is the database (conforming the relational metamodel in Figure 2) that is expected to be produced by the application of the input transformation on the input model. Figure 7 illustrates this implementation schema.

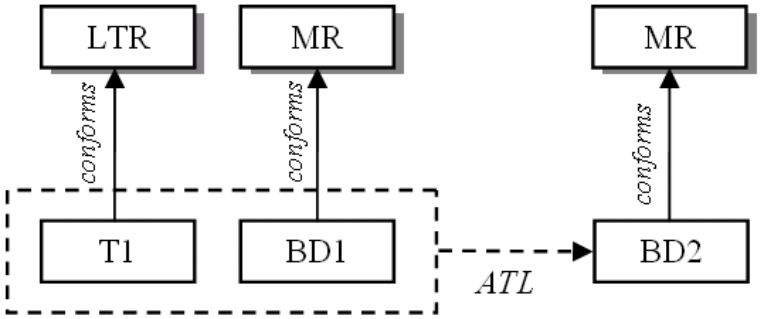



**Figure 7.** DSTL implementation schema using ATL transformations

In our implementation we use the ATL's refinement facility in order to simplify the transformation algorithm. The refinement mechanism allows us to write code only for the part of the source model that is modified by the transformation, while the rest of the model is translated from source to target without any modification.

```
module MRandLTR2MR;
create OUT : MR refining  IN : MR, T: LTR;
```

Each syntactic construct of the DSTL is implemented by one or more ATL transformation rules. The simplest construct named *ChangeName* is implemented by a single transformation rule, as follows:

```
rule ChangeName_table {
from
t1: MR!Table (not t1.getChangeName().oclIsUndefined()
to
t2: MR!Table (
        name <- t1.getChangeName().newName,
    element <- t1.element,
    primaryKey <- t1.primaryKey
    )
}
```

Notice that we have overcome the limitation of not being able to match more than one element at the same time by using helper functions. We have defined three helper functions that allow us to distinguish whether each selected element must be processed or not. The implementation of one of such functions is:

```
helper context MR!Table def: getChangeName(): LTR!ChangeName
  = LTR!ChangeName.allInstances()->select(t|t.table = self.name).first();
```

Next, we introduce the implementation of the *extractCommonData* construct. This construct is implemented by three transformation rules, each rule works in each level of the relational model (i.e., model, scripts and data values).

- The following rule realizes the transformation on the data model:

  The rule transforms the selected column to a foreign reference to the target table. The creation of the target table is considered in the imperative part of the rule.

```
rule ExtractCommonData_table {
from
    c: MR!Column (not c.getExtractCommonData().oclIsUndefined()
using {
t : LTR!ExtractCommonData = c.getExtractCommonData();
}
to
    fk: MR!ForeignKey (
        table <- c.table,
        name <- 'fk_' + t.TargetTable
    )
do {
    if   fk.table.bd.getTable(t.TargetTable).oclIsUndefined()
    then
        thisModule.NewExtractionTable(t.TargetTable,
                                       t.column,
                                       fk.table.bd)
     else true
    endif;
    fk.referencedTable <-
         fk.table.bd.getTable(t.TargetTable);
    }
}
```

- The following rule implements the transformation on the scripts:

The rule transforms the direct references to the extracted column, by an indirect reference to the column (not primary key) of the new table.

```
rule ExtractCommonData_script {
from
    r1: MR!DirectReferenceToElementTable
        (not r1.getExtractCommonData().oclIsUndefined())
using {
    t: LTR!ExtractCommonData = r1.getExtractCommonData();
}
to
    r2: MR!DirectReferenceToElementTable (
        name <- r1.name,
        element <- r1.element,
        reference <- ref
    ),
    ref: MR!DirectReferenceToElementTable (
    )
do {
    ref.element <-
    r2.element.table.bd.getTable(t.TargetTable)
    .getElementWithName(t.column);
}
}
```

- The following rule defines the transformation on the data values:

```
rule ExtractCommonData_data {
from
    d1: MR!ValueElementTable
    (not d1.getExtractCommonData().oclIsUndefined())
using {
    t: LTR!ExtractCommonData = d1.getExtractCommonData();
}
to
    d2: MR!ValueElementTable (
        data <- d1.data,
        element <- d1.element
    )
  do {
     if
        d2.element.table.bd.getRefData(t.TargetTable,
                    t.column, d1.value).oclIsUndefined()
     then
         thisModule.NewDataRef(
         d2.element.table.bd.getTable(t.TargetTable),d1.value)
     else
         true
     endif;

     d2.value <- d2.element.table.bd.getRefData(t.TargetTable,
                       t.column,d1.value).data.getValueId()
  }
  }
```

The rule above moves each data in the source column to the target table. The rule also specifies that these values are replaced by the corresponding values of the primary key in the new table. Finally, the implementation of the *factorize* construct is similar to the previous implementations and it is not presented here for space limitations. The complete implementation of this relational DSTL can be downloaded from http://sol.info.unlp.edu.ar/eclipse.

## 5 An alternative implementation approach

In order to show a wider range of alternatives, in this section we carry out the DSL implementation using a slightly different approach. A MOFScript transformation [Oldevik, 2006] takes the constructs of the relational transformation language as input and generates an ATL transformation. This means, we implement a translation from the domain specific transformation language to the general purpose transformation language. The translation rules are written in the model-to-text transformation language MOFScript. The generated ATL program can be seen as the semantics interpretation of our DSTL.

Figure 8 explains this translational approach for the definition of the semantics of the domain specific transformation language.

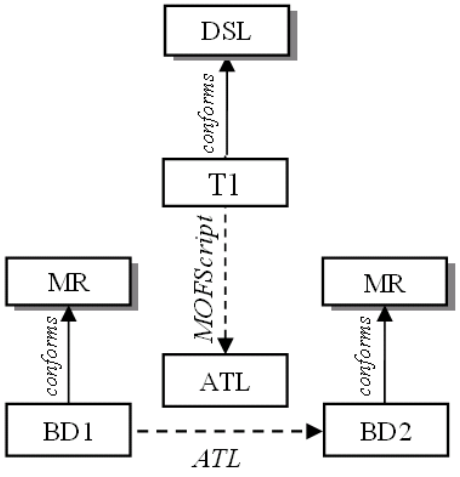



**Figure 8.** DSTL implementation schema using a translational approach.

In order to transform the DSTL sentences using MOFScript, the metamodel describing its abstract syntax must have a parent element (root). Figure 9 shows the customized metamodel for our relational transformation language.

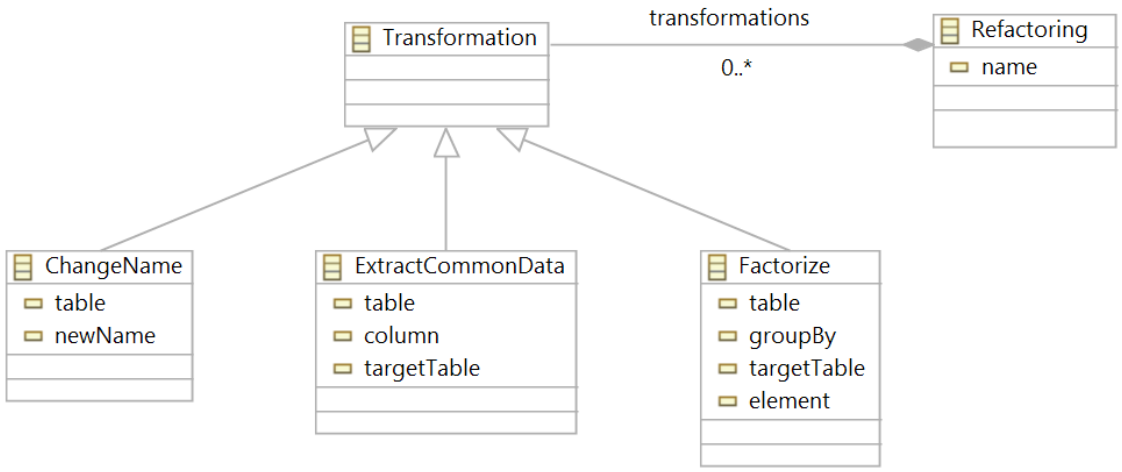



**Figure 9.** customized metamodel of the domain specific transformation language

The MOFScript transformation creates an ATL file, and each DSL sentence is translated by the application of a separate transformation rule.

```
texttransformation LTR2ATL (in ltr:"ltr") {

 ltr.Refactoring::main () {
   file(self.name + '.atl')

   '
   module ' self.name ';
   create OUT : MR refining  IN : MR;

   helper context MR!TableElement def : isPrimaryKey(): Boolean =
                                        self = self.table.primaryKey.element;

   '

   self.transformations->forEach(t:ltr.Transformation) {
       t.mapTransformation();
   }
 }
```

A rule named *mapTransformation()* is defined for each DSTL sentence. The following listing shows the implementation of these rules.

```
ltr.Factorize::mapTransformation() {
  '
  rule Factorize_' self.table '{
  from
      t1: MR!Table ( t1.name = \'' self.table '\')
  using {
    extraidos : Sequence(String) = ' self.element.mapSequence() ';
    elems : Sequence(MR!TableElement)
        = t1.element->reject(e| extraidos->includes(e.name) or e.isPrimaryKey());
    eprim : MR!Elemento = t1.element->select(e| e.name = \'' self.groupBy'\').first();
    elemsExtraidos : Sequence(MR!TableElement)
        = t1.element->select(e| extraidos->includes(e.name));
  }
  to
      t2: MR!Table (
          name <- t1.name,
          bd <- t1.bd,
          element <- elems
      ),
      cprim: MR!PrimaryKey (
          table <- t2,
          element <- eprim
      ),
      tfac: MR!Table (
          name <- \'' self.targetTable'\',
          bd <- t1.bd,
          element <- Sequence{id,elemsExtraidos,fkt2}
      ),
        id: MR!Column (
            name <- \'id_' self.targetTable '\',
            table <- tfac
        ),
        fkt2: MR!ForeignKey (
            name <- \'fk_\' + t1.name,
            table <- tfac,
            referencedTable <- t1
        ),
        cprimtfac: MR!PrimaryKey (
            table <- tfac,
            element <- id
        )
    }
    '
}
```

For example, given the following DSTL sentence:

```
<LTR:Refactoring name="RefactoringExample">
  <transformations xsi:type="LTR:Factorize" table="Documentos" groupBy="isbn" targetTable="Capitulos">
    <element>capitulo</element>
    <element>disponible</element>
  </transformations>
</LTR:Refactoring>
```

The ATL code generated by the application of the MOFScript transformation is:

```
module RefactoringExample;
create OUT : MR refining  IN : MR;

helper context MR!TableElement def : isPrimaryKey(): Boolean =
                                     self = self.table.primaryKey.element;

rule Factorize_Documentos{
from
    t1: MR!Table ( t1.name = 'Documentos')
using {
    extraidos : Sequence(String) = Sequence{'capitulo', 'disponible'};
    elems : Sequence(MR!TableElement)
        = t1.element->reject(e| extraidos->includes(e.name) or e.isPrimaryKey());
    eprim : MR!Elemento = t1.element->select(e| e.name = 'isbn').first();
    elemsExtraidos : Sequence(MR!TableElement)
             = t1.element->select(e| extraidos->includes(e.name));
 }
to
    t2: MR!Table (
        name <- t1.name,
        bd <- t1.bd,
        element <- elems
    ),
    cprim: MR!PrimaryKey (
        table <- t2,
        element <- eprim
    ),
    tfac: MR!Table (
        name <- 'Capitulos',
        bd <- t1.bd,
        element <- Sequence{id,elemsExtraidos,fkt2}
    ),
    id: MR!Column (
        name <- 'id_Capitulos',
        table <- tfac
    ),
    fkt2: MR!ForeignKey (
        name <- 'fk_' + t1.name,
        table <- tfac,
        referencedTable <- t1
    ),
    cprimtfac: MR!PrimaryKey (
        table <- tfac,
        element <- id
    )
}
```

The target ATL transformation takes a relational model as input and generates the transformed model, according to the semantics of the source relational transformation.

Although both implementation approaches look quite similar, there are differences and both have pros and cons. The disadvantage of the last schema lies in the difficulty of writing a transformation to generate an ATL transformation, where in fact we need two transformation steps. On the contrary, within the former scheme, the ATL translation is written directly, however this single transformation is much more complex.

# 6 Related work

There are a number of features of our work that can be contrasted to other current approaches:

**Abstraction and modularization of model transformations:**

Transformations are used more frequently, leading to the creation of increasingly bigger model transformation scripts. Our approach can be seen as a technique for transformation abstraction and modularization in that each high level transformation (written in the DSTL) is associated with a lower level transformation (written in GPTL), but the users do not need to be aware of the details of the low level transformation. In this sense, those approaches that propose techniques to build complex transformations by composing smaller transformation units are related to our proposal. In this category we can mention the composition technique described by Kleppe [2006], the Model Bus approach [Blanc et al., 2004], the modeling framework for compound transformations defined by Oldevik [2005] and the module superimposition technique described by Wagelaar [2008], among others. In contrast to these approaches, our proposal generates the composed transformation specification in a more simple way, without introducing any explicit composition machinery.

**Creating languages that abstract out from other more abstract languages:**

This subject has been intensely discussed in the literature on DSLs. For example, the MetaBorg [Bravenboer and Visser, 2004] is a transformation-based approach for the definition of embedded textual DSLs implemented based on the Stratego framework. Similarly to our work, the MetaBorg approach defines new concepts (comparable to our notion of an abstract language) by mapping them to expansions in the host language (comparable to our notion of a concrete language). The work in [Johannes et al., 2009] shows how to develop DSLs as abstractions of other DSLs by transferring translational approaches for textual DSLs into the domain of modeling languages. The underlying notion of an embedded DSL seems to have been discussed first by Hudak [1998]. The idea of forwarding has been introduced in [Van Wyk et al., 2002]. An important distinction between these works and our proposal is the application to the model transformation field.

**Concrete-syntax-based transformations:**

Contrary to traditional approaches to model transformation, the work presented in [Baar and Whittle, 2007], uses the concrete syntax of a language for expressing transformation rules, which is very similar to our proposal. They claim that this simplifies the development of model transformations, as transformation designers do not need deep knowledge of the language's metamodel. In our approach, we use the abstract DSTL with a similar purpose: users do not need to count with a deep knowledge of the abstract syntax of the involved modeling languages, but they just use the simple syntax of the DSTL.

# 7 Conclusions

We have presented a translational approach for defining abstract domain-specific transformation languages (DSTLs) based on concrete general purpose transformation languages (such as ATL).

In contrast to an approach where a general purpose transformation language is used, our approach provides the following benefits:

- The complexity of transformation programs gets reduced. A program is composed by few lines of high expressive commands.
- Domain experts will feel more comfortable using a specific language with constructs reflecting well-known concepts (such as, table and column in our example); consequently it is predictable that they will be able to write more understandable and reusable transformations in a shorter time.

- Transformation developers do not need to know the intricate details of model transformation languages, as these are encapsulated in the DSL constructs. This leads to a natural separation into a language designer and a transformation designer role, with a reduced learning effort for the later.

Additionally we propose that the semantics of such DSTL is defined using a transformation language itself. We have taken an approach where the DSTL instance is not compiled into source code but transformed onto a generic model transformation language. In this case we have used ATL. This fact provides several advantages:

- The language semantics is formally described, and it is executable;
- The semantics is understandable because it is written in a well-known language;
- The semantics can be easily modified. Although the ATL transformation may be considered as a compiler, the amount of programming skills required to create it is much lesser than for creating a compiler to source code.

As an experimental example in this paper we have reported the definition of a DSTL in the domain of databases and we have described its implementation in ATL. The experience was successful, showing the advantages of defining DSTL for model refactoring – i.e., transformations that locally change an existent model producing a new model that conforms to the same metamodel. Currently we are working in the definition of other DSTL in other domains.

It is also important to take into account the benefits coming from the platform-independence of the transformation language: we are able to transform and execute its instances onto different generic transformation language platforms – e.g., we may use QVT instead of ATL.

## Acknowledgments

This work has been sponsored by Microsoft® under the LACCIR RFP 2008 Research Founding Initiative.

## References

ATLAS team (2006). ATLAS MegaModel Management (AM3), Home page: http://www.eclipse.org/gmt/am3/.

Baar, T., and Whittle, J. (2007). On the Usage of Concrete Syntax in Model Transformation Rules. In Book: Perspectives of Systems Informatics. LNCS 4378, Springer Heidelberg, Berlin.

Barbero, M., Bézivin, J., and Jouault, F. (2007). Building a DSL for Interactive TV Applications with AMMA. In TOOLS Europe'07: Proceedings of the Workshop on Model-Driven Development Tool Implementers Forum. June. Zurich, Switzerland.

Bézivin, J., Jouault, F., Kurtev, I., and Valduriez, P. (2006). Model-based DSL Frameworks. OOPSLA Companion'06, pp. 602–616.

Blanc, X., Gervais, M., Lamari, M. and Sriplakich, P. (2004). Towards an integrated transformation environment (ITE) for model driven development (MDD). In SCI'04: Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics. July. USA.

Bravenboer, M., and Visser, E. (2004). Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In OOPSLA'04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press. pp. 365–383.

Cook, S., Jones, G., Kent, S., and Wills, A. (2007). Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional. ISBN 0-321-39820-3.

Czarnecki, K., and Helsen, S. (2006). Feature-based survey of model transformation approaches. IBM System Journal, 45(3): 621–645. July.

Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., and Pierantonio, A. (2009): Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Downloaded March: http://hal.ccsd.cnrs.fr/docs/00/06/61/21/PDF/rr0602.pdf

Greenfield, J., Short, K., Cook, S., and Kent, S. (2004). Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley.

Gronback, R. (2009). Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional. ISBN: 0-321-53407-7.

Hudak, P. (1998). Modular domain specific languages and tools. In ICSR'98: Proceedings of the 5th International Conference on Software Reuse, IEEE Computer Society Press. pp. 134–142. June. Victoria, B.C., Canada.

ISIS-GME (2008). GME: The Generic Modeling Environment. ISIS Institute, School of Engineering, Vanderbilt University, Nashville, TN, USA. Reference site: http://www.isis.vanderbilt.edu/Projects/gme

Johannes, J., Zschaler, S., Fernandez, M., Castillo, A., Kolovos, D., and Paige, R. (2009). Abstracting Complex Languages through Transformation and Composition. In MoDELS'09: Proceedings of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems. USA, LNCS, Springer. October. Denver, Colorado, USA.

Jouault, F., and Kurtev, I. (2006). Transforming Models with ATL. In Proceedings of Satellite Events at the MoDELS 2005 Conference. LNCS 3844, Springer-Verlag, pp. 128–138.

Jouault, F., Bézivin, J., Consel, C., Kurtev, I., and Latry, F. (2006). Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In ECOOP'06: Proceedings of the 1st Workshop on Domain-Specific Program Development (DSPD), July. Nantes, France.

Kleppe, A. (2006). MCC: A Model Transformation Environment. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, Springer-Verlag, pp. 173–187, June. Spain.

Kleppe, A., Warmer, J., and Bast, W. (2003). MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Mernik, M., Heering, J., and Sloane, A. (2005). When and how to develop domain-specific languages. ACM Computing Surveys, 37(4):316–344.

Oldevik, J. (2005). Transformation Composition Modeling Framework. In DAIS'05: Proceedings of the 5th IFIP International Conference on Distributed Applications and Interoperable Systems. LNCS 3543, Springer-Verlag, pp. 108–114, June. Athens, Greece.

Oldevik, J. (2006). MOFScript User Guide. Version 0.6 (MOFScript v 1.1.11).

OMG/MOF (2003). Meta Object Facility (MOF) 2.0. OMG Adopted Specification. October. http://www.omg.org

OMG/QVT (2005). MOF QVT Adopted Specification 2.0. OMG Adopted Specification. November. http://www.omg.org

Pons, C., Giandini, R., and Pérez, G. (2010). Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica. Editorial: EDUNLP and McGraw-Hill Education.

Sánchez Cuadrado, J., García Molina, J., and Menarguez Tortosa, M. (2006). RubyTL: A Practical, Extensible Transformation Language. In Proceedings of European Conference on Model Driven Architecture – Foundations and Applications, LNCS 4066. Springer-Verlag.

Stahl, T., and Völter, M. (2006). Model-Driven Software Development. John Wiley & Sons, Ltd.

Van Wyk, E., de Moor, O., Backhouse, K., and Kwiatkowski, P. (2002). Forwarding in attribute grammars for modular language design. In Horspool, R.N., ed.: Int Conf. on Compiler Construction. LNCS 2304, Springer, Berlin / Heidelberg pp. 128–142.

Wagelaar, D. (2008). Composition Techniques for Rule-based Model Transformation Languages. In ICMT'08: Proceedings of the International Conference on Model Transformation. July. Zurich, Switzerland.