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

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

[1] LIFIA, Facultad de Informática, Universidad Nacional de La Plata
[2] CONICET, Consejo Nacional de Investigaciones Científicas y Técnica
[3] Universidad Abierta Interamericana (UAI)
Buenos Aires, Argentina
[jirazabal,cpons]@lifia.info.unlp.edu.ar

Abstract. Model Driven Engineering proposes a software development process in which the key notions are models and model transformations. 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. The term "model transformation language" comprises all sorts of artificial languages used in model transformation development such as QVT, ATL and RubyTL. These languages are specific to define model transformations; however an extra level of specialization can be realized on them. That is to say, we can define a transformation language specifically addressed to a given transformation domain. In the present work we introduce the proposal of defining domain specific transformation languages and also we analyze a novel way to define their 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 data base 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) [1] [2] [3] proposes a software development process in which the key notions are models and model transformations. 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. Model transformation is the MDE engine.

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 [4]. 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) [5], modeling and meta-modeling languages and ontologies. Examples include languages such as the standard QVT [6], ATL [7] [8] and RubyTL [9].

These languages are specific to define model transformations; however an extra level of specialization can be realized on them. That is to say, we can define a transformation language specifically addressed to a given transformation domain. For example, we can create a language dedicated to the definition of transformations between data-base models or a language addressed to the definition of transformations between business models. 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 exist powerful frameworks for the definition of domain specific languages, such as Eclipse [10][11], Microsoft DSL Tools [12][13] and AMMA [14]. 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. The AMMA framework is the exception; it takes advantage of the MDE ideas. Within AMMA the semantics of the DSL can be defined more abstractly in terms of Abstract State Machines (ASMs) or in terms of another language. In [15], the application of this framework to the implementation of the languages SPL and CPL is described.

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 implement 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 implementation of such DSTL. Section 5 compares this approach with related research and presents the conclusions.

2 DSL implementation schema

The AMMA framework [14] allows us to define the concrete syntax, abstract syntax, and semantics of DSLs. In [15, 16, 17] 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 is similar to the one of AMMA, but we present a novel alternative, where the language semantics is realized 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 (that is to say, a domain specific transformation written in the domain specific language, for example a transformation between data bases) and a model belonging to the specific domain (for example, a data base 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.

Figure 1 shows the transformation scenario.

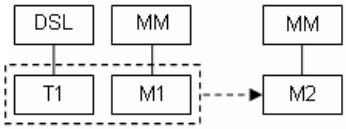


Figure 1. Transformation scenario.

In our implementation we directly deal with the abstract syntax of the DSL. This simplification can be easily relaxed in order to also consider concrete syntaxes; for example we could use the TCS language which is provided by AMMA to this 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 data base 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 data base model.

3.1 The relational model

Due to the fact that the transformation language is expected to express the transformation of the whole spectrum of the data base (i.e., the data model, the scripts and the 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, as described in [6], that is restricted to the M1 level of the OMG 4-levels metamodeling architecture [18]; that is to say, our metamodel contains additional meta-classes to represent scripts and data values as well. Figure 2 shows the upgraded relational metamodel.

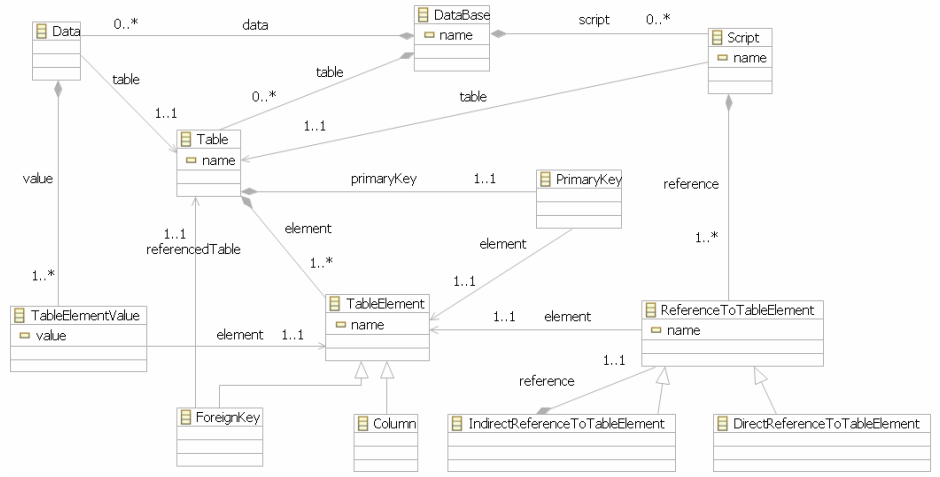


Figure 2. Simplified relational metamodel including scripts and data values

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 data bases. 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 data bases domain. As 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.

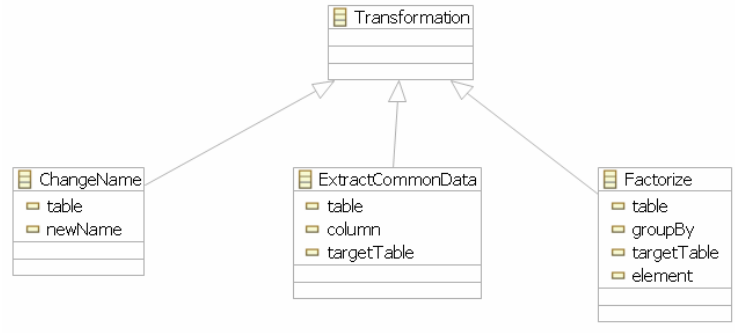


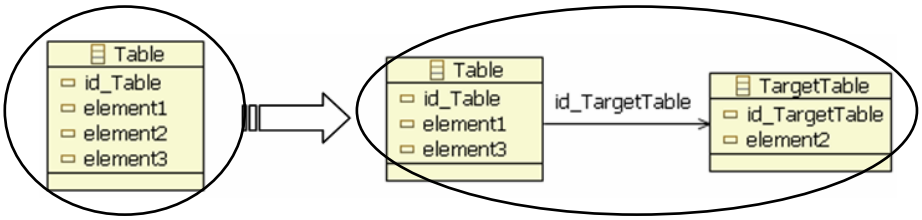
Figure 3. Metamodel of the domain specific transformation language

After having defined the syntax of our language we need to define its semantics. In first place we describe the semantics using just natural language. These definitions 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.

– ***changeName***: this is a very simple transformation, its effect consists in 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"/>
```

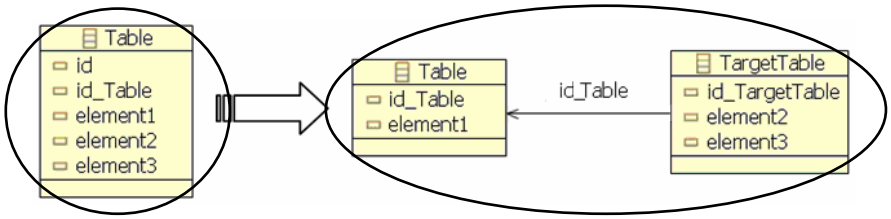
Figure 4. Effect of the *extractCommonData* transformation**.**

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 target 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 foreign key to the new table as the value of the primary key of the new table, corresponding to the value of each data in the source table

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 it is expected, the evaluation of any transformed script on the transformed data base will present no observable difference with respect to the evaluation of the corresponding source script on the source data base.
The effect of the transformation on the data model is illustrated in figure 5.




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

Figure 5. Effect of the *factorize* transformation.

Thinking algorithmically, 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.

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

3.3 Example

In this section we show the applicability of the domain specific transformation language. To this purpose we elaborate a very simple example consisting of a simple data base containing a single table named Book. The table has seven columns: ISBN, title, editorial, comments, availability, chapterTitle and chapterPages.

By using our DSTL we will transform this data base to a behavioral equivalent data base without data duplication.

To specify the transformation we use a concrete syntax based on XML and directly supported by AMMA, as follows:

```
<ExtractCommonData table="Book" column="editorial"
TargetTable="Editorial"/>
<FactorizeTable table="Book" groupBy="isbn"
TargetTable="Chapter">
```

Figure 6 displays the source model on the left hand and the target model (the result of the transformation) on the right hand.

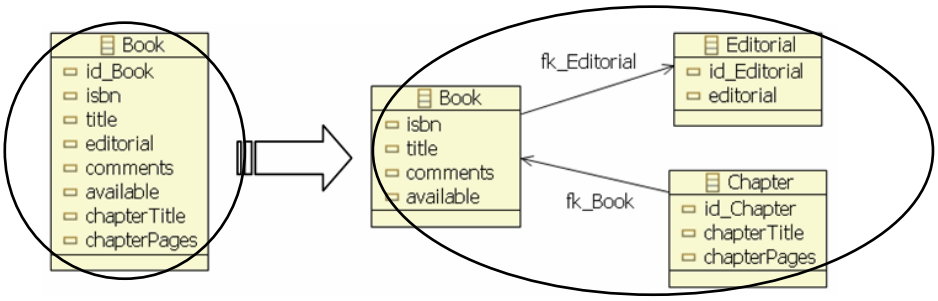


Figure 3. The data model before and after the transformation application.

After applying the first transformation, the editorial info is not longer a column in the table. The editorial info becomes an entity in the target data base. 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.

4 DSTL Implementation

In this section we present the implementation of our DSTL by using the model transformation language ATL. The implementation consists in a transformation, written in ATL that takes two inputs: a relational data base (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 data base (conforming the relational metamodel in figure 2) that is expected to be produced by the application of the input transformation on the input model.

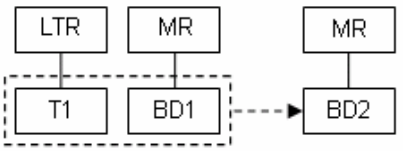


Figure 4. DSTL implementation schema using ATL transformations

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
    )
}
```

Como pudo observarse, hemos resuelto la limitación de no poder machear más de un element en simultáneo utilizando funciones auxiliares. Tendremos tres funciones auxiliares que nos permitirán saber si el element macheado debe ser transformado o no. La siguiente es la implementación de una de las tres funciones utilizadas:

```
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:

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

– The following rule implements the transformation on the scripts:

```
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).get
      ElementWithName(t.column);
}
}
```

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

– 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()
}
}
```

This rule moves each data in the source column to the target table; the rule 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 relatonal DSTL can be downloaded from http://sol.info.unlp.edu.ar/eclipse.

5 Conclusions and related work

Our proposal of using domain specific transformation languages instead of general purpose transformation languages (such as ATL) is expected to reduce the complexity of transformation programs. 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.

Additionally we propose the semantics of such DSTL to be defined using a transformation language itself (i.e., ATL). This fact provides several advantages: the language semantics is formally described; it is executable; the semantics is understandable because it is written in a well-known language; the semantics can be easily modified.

As an experimental example in this paper we have reported the definition of a DSTL in the domain of data bases and we have described its implementation in ATL. The experience was successful; currently we are working in the definition of other DSTL in other domains.

Acknowledgments

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

6 References

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

[2] Claudia Pons, Roxana Giandini, Gabriela Pérez. "Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica". Editorial: EDUNLP and McGraw-Hill Education. (2010).

[3] Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (2003)

[4] Czarnecki, Helsen. Feature-based survey of model transformation approaches. IBM System Journal, V.45, N3, 2006.

[5] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. ACM Computing Surveys, 37(4):316–344, 2005.

[6] MOF QVT Adopted Specification 2.0. OMG Adopted Specification. November 2005. http://www.omg.org

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

[8] 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

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

[10] GME: The Generic Modeling Environment, Reference site, http://www.isis.vanderbilt.edu/Projects/gme. (2006).

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

[12] Steve Cook, Gareth Jones, Stuart Kent, Alan Cameron Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional. ISBN 0321398203, 2007.

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

[14] Bézivin, J., Jouault, F., Kurtev, I., Valduriez, P.: Model-based DSL Frameworks. (2006) OOPSLA Companion 2006:602-616[5].

[15] Frédéric Jouault, Jean Bézivin, Charles Consel, Ivan Kurtev, Fabien Latry: Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD), July 3rd, Nantes, France (2006).

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

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

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