Towards a Unified Notation to Represent Model Transformation

Anne Etien, Cedric Dumoulin, Emmanuel Renaux

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards a Unified Notation to Represent Model Transformation

Anne Etien — Cedric Dumoulin — Emmanuel Renaux —
Email: {etien, dumoulin, renaux}@lifl.fr

N° ????

May 9, 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

Towards a Unified Notation to Represent Model Transformation

Anne Etien , Cedric Dumoulin , Emmanuel Renaux ,
Email: {etien, dumoulin, renaux}@lifl.fr

Thème COM — Systèmes communicants
Projet DaRT

Rapport de recherche n° ???? — May 9, 2007 — 14 pages

Abstract: In order to unify our internal exchange and communication about transformations, we propose TrML (Transformation Modeling Language), a unified UML notation to design model transformations. This proposal aims to reify the synthesis of existing notations dedicated to transformation modeling. TrML is independent from implementation details and could be adapted to several transformation engines. To let TrML run on top of existing engine, we transform TrML model to a model accepted by the engine. But, which language should we use for the first transformation? TrML, the targeted engine or another one? In this article we will describe how we bootstrap our new language on top of existing transformation engines.

Key-words: Model transformation, Bootstrap, TrML

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 60 19 66 08

Vers une notation unifiée pour modéliser les transformations de modèles

Résumé : Afin d'uniformiser au sein de l'équipe les échanges et la communication sur les transformations de modèles, nous proposons TrML (Transformation Modeling Language), une notation unifiée pour représenter graphiquement les transformations de modèles. Cette proposition a pour but de concrétiser la synthèse de notations existantes dédiées aux transformations de modèles. TrML est indépendant des détails d'implémentation et peut être adapté pour différents moteurs de transformation. Afin d'utiliser TrML avec des moteurs existant, nous transformons le modèle TrML vers un modèle accepté par le moteur choisi. Maix quel langage doit-on choisir pour écrire la première transformation ? TrML, le modèle cible adapté au moteur ou un autre langage ? Dans cet article, nous décrivons comment 'bootstraper' notre langage au dessus de moteurs de transformation existants.

Mots-clés : Transformation de modèles, Bootstrap, TrML

Introduction

Transformation chains involve several models from high abstraction level to implementation levels. Each abstraction level refers to a specific system viewpoint that an expert is responsible for. Thus to build the whole product chain, experts have to exchange and communicate about transformations. They need an abstract language focusing on the intention of the transformation and that homogenizes documentation.

Since years, modeling activities have been mainly supported by UML-like modeler tools and then, visual notation has become more familiar. Graphical representation of transformation model thus seems to be the best abstraction to focus on the purpose of the transformation independently from the implementation details. Some graphical transformation languages propose a notation visually close to classical representation of the model the transformation applies to. Most of these contributions emerge from graph community [5, 2] and not from the Model Driven Engineering domain that however recommends separation of graphical notation from the implementation language.

We planned empirical studies about the need of a visual notation to represent model transformation independently from implementation platform. After having assessed existing model transformation solutions, we have developed the Transformation Modeling Language TrML. In this modeling language, we gather the core features that we consider mandatory to exchange and communicate about transformation. We provide the corresponding UML profile that makes it portable on any UML tool. TrML is not associated to a transformation engine but is defined in a metamodel allowing to map it on any languages.

Transformations aim to be automatically executed. It is thus essential to provide a transformation engine for TrML transformations. We could have constructed our own transformation engine, but it is contradictory to the essence of TrML that aims to provide a graphical notation independently from the transformation engine or language. Furthermore, there are several transformation engines that have proved their efficiency. We thus propose to build TrML on top of an existing transformation engine. For this purpose, we adopt the bootstrap mechanism allowing to write a transformation from TrML to the model associated to the targeted engine. ATL has been chosen as the targeted engine. However, the principles described in this paper are general and can be applied to other languages such as for example QVT.

The paper is organized as follow. Section 2 describes the basic features of TrML illustrated with the UML to RDBMS example. Section 3 presents the implementation of TrML on top of an existing transformation language and illustrates it with ATL. Section 4 compares TrML to other works on model transformation. Section 5 concludes the article.

1 TrML to graphically design model transformations

This section details TrML core features and illustrates the syntax and the basic semantics of the design representation, specifying UML2RDBMS transformation example. This example is extracted from the QVT specification and describes a simple transformation from a class model to a database schema, summed up in the following sentences:

- *s1. a persistent class maps to a table, a primary key and an identifying column.*
- *s2. attributes of the persistent class map to columns of the table:*
  - *an attribute of a primitive data type maps to a single column;*
  - *an attribute of a complex data type maps to a set of columns corresponding to its exploded set of primitive data type attributes;*
  - *attributes inherited from the class hierarchy are also mapped to the columns of the table.*

- *s3. an association between two persistent classes maps to a foreign key relationship between the corresponding tables.*

1.1 Description of TrML features through an example

Rule. Transformations are mostly too complex to be performed on one shot. They are decomposed into a set of rules that are more legible and allow to reduce the problem to solve. A rule focuses on a small part of the models to be transformed. It is made of input and output patterns describing what should be transformed into what, and how elements are linked (via bind names).

There are different kind of rules: toprules (Figure 1), that are directly executed by the engine; normal rule (Figure 3), explicitly called from another rule; setRules (Figure 2) and listRules, are collections of rules where each rule is executed in any order or in the specified order respectively; selectRule, an ordered set of rules where only the first applicable rule is executed.

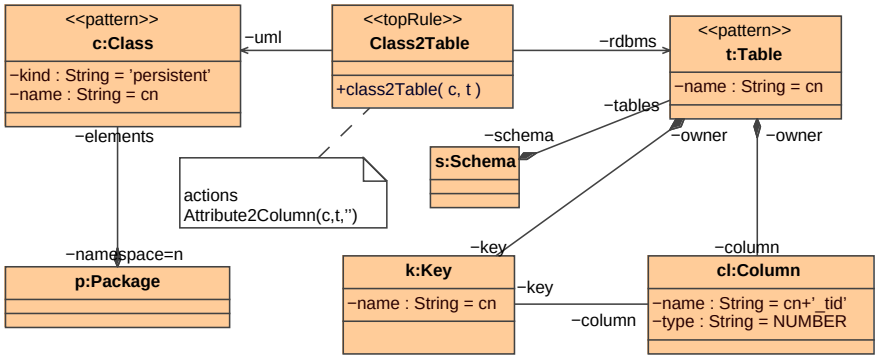

Figure 1: Description of the Class2Table rule

One goal of TrML is to be able to describe a rule in one diagram. We usually dispose source patterns on one side, and target patterns on the other side. Each pattern is layed in a graph manner, starting from the root stereotyped ≪pattern≫. Also, we use notes instead of other diagrams or UML artefacts in order to describe additional informations such as rule calls, constraints, small computation... We could imagine that such notes are generated from more appropriate UML constructs (activity diagrams, action language...), but for now we really want to be able to understand a rule in one glance.

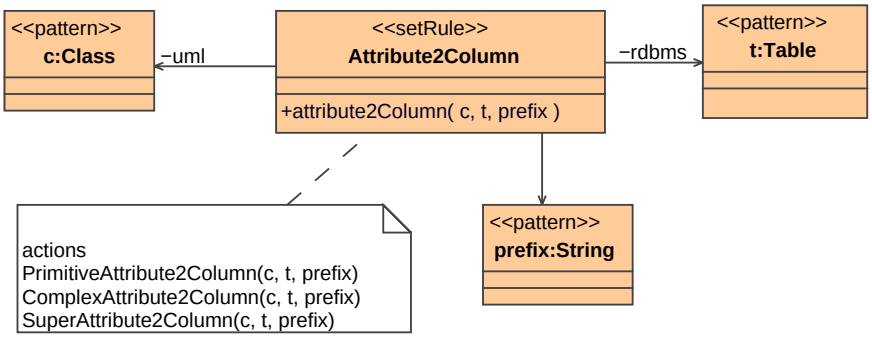

Figure 2: Description of the Attribute2Column rule

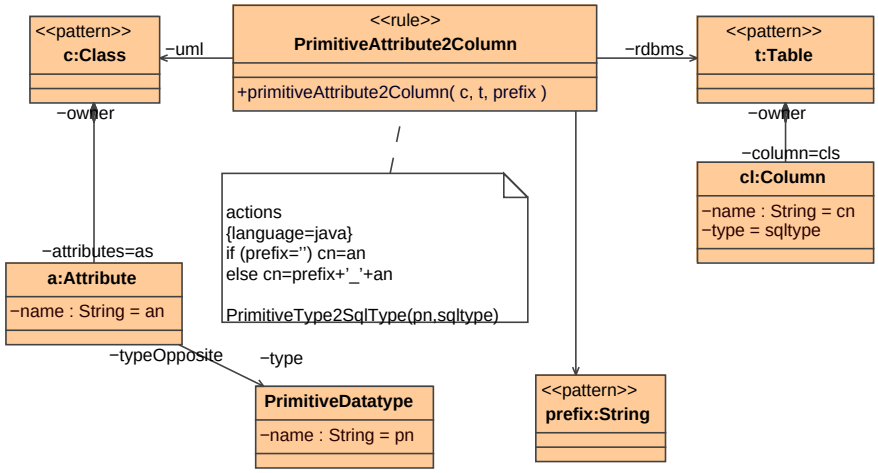

Figure 3: Description of the PrimitiveAttribute2Column rule

Pattern. TrML proposes to describe elements involved in the rules into two sets: the source patterns and the target patterns. The source patterns describe a part in the input model while the target patterns describe what should be generated from the input patterns.

A pattern uses a UML object like notation to describe only the features necessary to execute the rule, i.e. attributes, operations and associations. Figure 4 presents the rule ComplexAttribute2Column that transforms the complex Attributes of a Class by calling Attribute2Column rule according the second point of the s2 sentence. The source pattern describes a Class that has Attributes which type is another Class of the UML model.

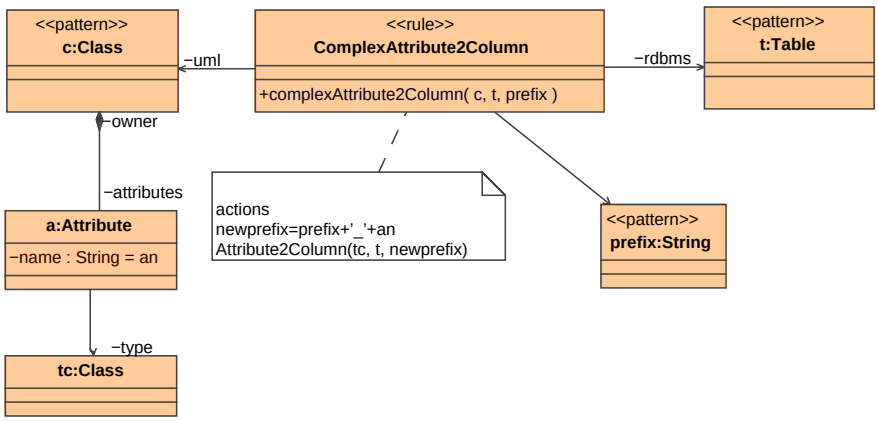

Figure 4: Description of the ComplexAttribute2Column rule

The target pattern of the Class2Table rule (see Figure 1) describes a Table which belongs to a Schema and which has got a Column corresponding to the Key of this table. The Table, the Column and the Key elements have a name. The Column besides has a type.

A pattern can also correspond to parameters when they are not linked to a model like in the Attribute2Column rule. The third pattern corresponds to the prefix parameter.

Bind name. Bind names are associated to pattern elements to manipulate constructs within patterns of the rules. They also allow to specify links between source pattern elements and target pattern

elements. Associated to classifiers or association ends, they describe precisely the rule behavior. In the Class2Table rule (see Figure 1), the bind name *c* manipulates the UML Class and *n* helps to use the namespace referring to the Package of the Class.

Associated to features, bind names allow to hold value from input patterns to output patterns. In Figure 1, the bind name *cn* specifies that the Table resulting from the transformation has the same name as the UML Class. In the source pattern, the expression *name:String=cn* allows to affect the value of the Class name to the *cn* bind name. In the target pattern, the expression *name:String=cn* affects this value to the name of the Table. Expression *name:String=cn+'_tid'* specifies that the name of the Column corresponds to the name of the Class suffixed by the key word *_tid*.

Guard. A guard is a condition applied to pattern elements that restricts the application domain of a rule. In the Class2Table rule (see Figure 1), the expression *kind:String = 'persistent'* is a guard as it restricts the application of the rule only to persistent classes. Here again, in order to visualize the rule on one shot, guard conditions are mostly specified in a note artifact. When the condition expression is simple, the default value field of the attributes can be used. The guard conditions may be separately defined in a formal language as OCL or UML action language.

Action. An action represents some additional computation that should be performed by the rule, like rule call, script execution or external call. Actions are usually described in a note in order to keep all the rule information on one single diagram.

Rule call. This is the way of calling sub-rules. For example, the SuperAttribute2Column rule in 5 looks for attributes of the super classes and delegates to the Attribute2Column rule the transformation of these attributes (as described in the last point of the s2 sentence).

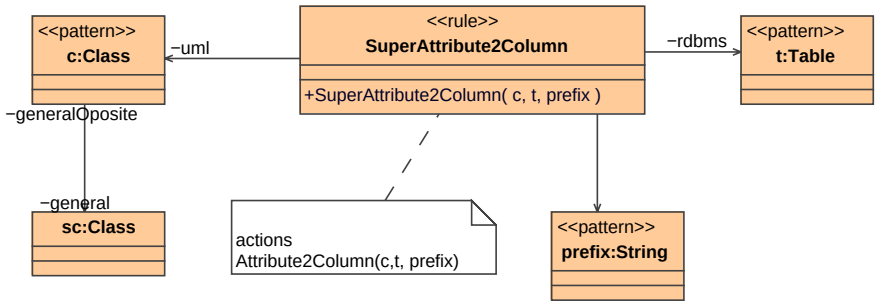

Figure 5: Description of the SuperAttribute2Column rule

From the same way, the Class2Table rule in **??** is complex and must be decomposed into other rules. The note attached to the rule contains a rule call. The parameters (*c* and *t*) specify the elements of the source and target patterns, the sub-rule is applied on. Thus, the sub-rule Attribute2Column(c,t) transforms the Attributes of the Class *c* into Columns of the Table *t*. The rule call improves the readability and the reuse of the Class2Table rule.

Script action. Some parts of the transformation can not be written with rules, for example because decomposition into patterns is not adapted. For these cases, some engines propose to write imperative code. TrML support this kind of code by the way of scripts. A script refers to a piece of code in a language, chosen by the designer or adapted to the transformation engine and explicitly specified in the rule. A script depends on the rule context and thus can use the bind names defined in the rule. When a script is simple, it can be written in the default value field. However, when the expression is more complex than adding a prefix or a suffix, for example by including application conditions, it is better to use actions note. As for the guards, the note only provides a graphical notation of the action.

The PrimitiveAttribute2Column rule (see Figure 3) provides such an example. The action note attached to the Column element contains an expression that determines the name *cn* of a column depending on whether the variable *prefix* is empty or not.

External call action. TrML philosophy is to design a transformation using a simple graphical notation. However, transformation can include singular manipulations that are really difficult to represent visually or that already exist. In those cases, the external call allows to add a method whose only the inputs and outputs are known, and thus to consider it as a black box. This code is independent from the transformation context and can thus not use the bind names directly but through input or output parameters.

For example, the PrimitiveAttribute2Column rule in Figure 3, contains PrimitiveType2SQLType that allows to transform a UML primitive type *pn* into a SQL type *sqltype*. The way the type transformation occurs is entirely hidden to the users, only the input *pn* and output *sqltype* parameters are known. The rest of the rule explains where the input parameter *pn* comes from and how the output parameter *sqltype* is used in the target pattern.

Definition. A definition is a property or an operation that adds a property to a model or performs a computation on a source element to help in providing target product. A definition can be defined within a rule or be used by the whole transformation. For instance, a definition can help to reach data in the model by factorizing a complex query.

1.2 Formalization

Table 1 gathers the different stereotypes provided in the TrML profile. Furthermore, Two additional attributes are associated to the transformation stereotype:

- models: TrML::model corresponding to the different models manipulated by the transformation.
- defaultTarget: TrML::model indicating a reading direction.

Table 1: Stereotypes of the TrML profile

Stereotype	Metaclass	Designation
transformation	Model	characterize the transformation model.
model	Model	distinguish the original model from its reference used in the rule transformation.
rule	Class	enable to link the source and the target patterns.
topRule	Class	characterize the rules directly executed by the transformation engine.
selectRule, setRule, listRule	Class	call several rules in one shot
pattern	Class	in the set of model elements necessary to the rule execution, only the handle of the rule is stereotyped as pattern; the other elements are UML classes.

Figure 6 formally specifies the concepts used in TrML within a metamodel and presented in the previous section through the UML2RDBMS transformation example.

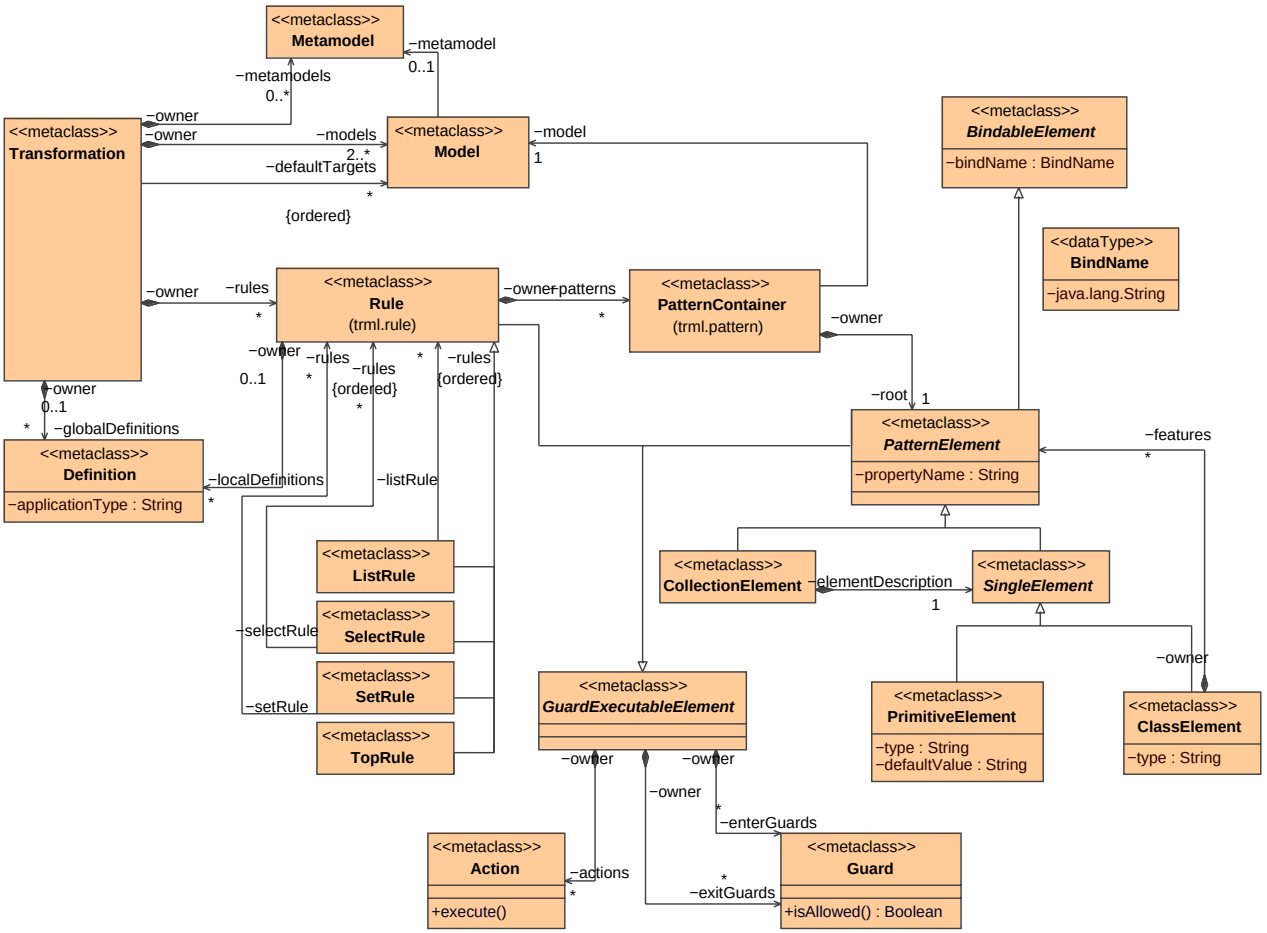

Figure 6: GaspardExcerpt of the TrML MM. blabla.

1.3 TrML control structures

TrML provides facilities to increase quality of the design in terms of reuse, factorization, and legibility. These facilities are associated to basic control structures such as the selection of the appropriate rules or the rule call. The use of these structures depends on the targeted transformation engine.

Control of the rules. TrML provides simple mechanisms to control the flow of rules execution. Each rule is responsible for calling the needed sub-rules in order to delegate part of its behavior or to reuse existing rules. Rules are thus layered, and topped by topRules that an engine can execute directly. The setRule, listRule and selectRule concepts take part in the control since they allow to gather rules into sets organizing their execution depending on respective policies.

The designer is thus responsible for the execution control of the complete transformation. He specifies simultaneously the intention and the organization of the rules, insofar as the control is defined within the rule and does not require an external mechanism.

Moreover smart references, defined below, provide an additional control mechanism that can solve rules inter-dependency relationships.

Smart references. An inner mechanism allows to keep trace of past rule execution or wait for needed new element creation by other rule. For example, in the PrimitiveAttribute2Column rule (Figure **??**, for each Attribute *a* of the Class *c*, one Column *cl* is added in the Table *t*, where t results

from the transformation of the Class *c* following the execution of the Class2Table rule. No new Table corresponding to the Class *c* is created. In the opposite, the smart references help to attach the new Column to the right Table. The smart references thus induce a part of the control, implicitly specifying which elements of the target pattern need to be previously transformed.

2 Implementation

TrML provides mechanisms to represent models transformations that have vocation to be automatically executed by an engine. There already exist several transformation engines that have proven their efficiency. We don't want to provide a new transformation engine. Instead, we want to be able to use them. Thus TrML provides a graphical way to design rules that will be executed on existing transformation engines.

From a user point of view, we want to write a transformation in the new language, and be able to execute it on the targeted engine. We don't care how the new language is executed by the targeted engine. From the targeted engine point of view the new language should be transformed to something understandable by the engine itself.

2.1 General considerations

The new brand transformation language has to be executed on top of an existing transformation engine (Figure 7). From a user point of view, we just want to write rules in the new language. We don't care how they are executed (the ovale shape in Figure 7). But, the targeted transformation engine only accepts its own language. So, we need to detail how the new language rules are executed. We propose to write some transformations from the new language to the language required by the targeted transformation engine (rules *new2target* in 7).

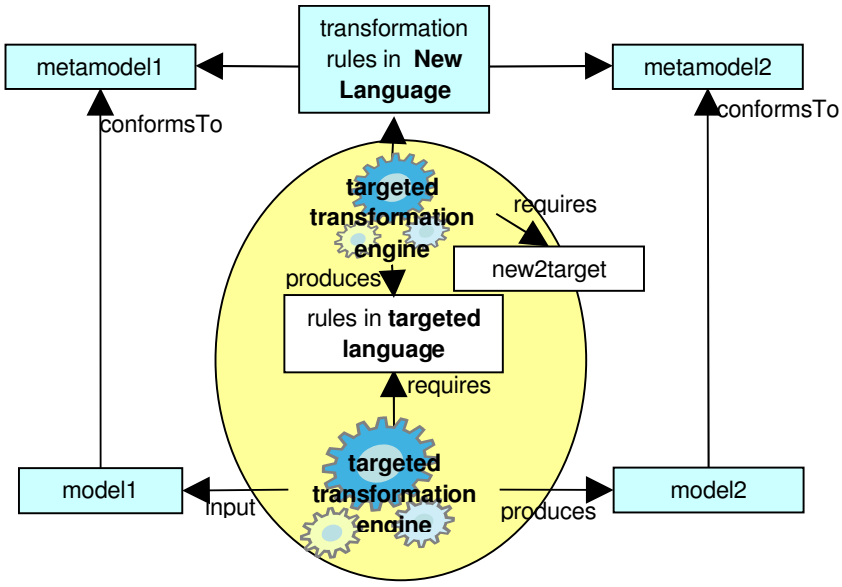

Figure 7: New Transformation Language executed on top of existing transformation engine.

In order to be able to execute a new language, some restrictions on the targeted language should be taken in consideration: there exist a transformation between the new language and the targeted engine language. In other words, the new language has to be automatically transformed to the targeted engine language.

The targeted engine accepts a model as rule descriptions. If not, there exists a code generation or whatsoever to transform a model of the new language to the required input of the targeted engine.

The question is: which language should be used to write *new2target*? The targeted language or the new language can be chosen. Let discuss on the advantage and drawback of each approach.

New Language - Ideally, it is the one we want to use. Otherwise, why boring to develop a new language? However, it cannot be used the very first time, as the transformation chain is not operational.

Targeted language - Writing the transformation with the targeted language will allow having an operational chain based on the targeted engine. The operational chain allows to obtain the code in the targeted language corresponding to a transformation written in the new language. Thus, transformation written in the new language can be executed without having to define a new engine.

This question concerning the way the transformation is performed leads to reflections close to those relative to the specification of a new C compiler written in C. A C compiler can be coded in C because it is a piece of code that can compile the C code and thus compile the new compiler. This bootstrap mechanism allows then to use the new compiler to compile new C code. We believe that the same bootstrap mechanism can be used to build a new transformation language on top of an existing transformation engine.

It is thus necessary to write a transformation from the new language to the targeted language with this latter. A transformation specified with the new language can thus be executed. The transformation from the new language to the targeted one can be written again but this time using the new language.

In the following section, we develop the bootstrap mechanism with the TrML profile as the new language and ATL as the targeted language.

2.2 Bootstrapping with ATL

In this section, we present how TrML can be added on top of the ATL engine. The demonstration is done with ATL [6], but it can be done with any other available engine. Just replace the name “ATL” by the name of your targeted engine.

The aim is to execute on the ATL engine a transformation written with the TrML profile, i.e. with UML. The UML model thus needs to be transformed to an ATL model. This transformation is done by a chain of transformations (Figure 8): the first transformation transforms the UML model to a TrML model; the second transformation transforms the TrML model to an ATL model. This later model can be executed on the ATL engine. In fact, ATL performs another transformation from the ATL model to ATL text and execute this later via an injector / extractor mechanism.

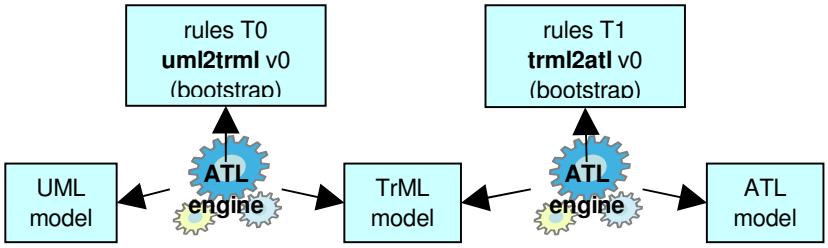

Figure 8: Transformation Chain to transform from new language (TrML-UML) to targeted engine language (ATL)

In the rest of the article a transformation chain corresponds to the set of transformation rules needed to go from UML to the targeted language and a transformation chain execution engine is the engine used to execute the transformation chain (i.e. ATL engine in Figure 8). The engine can execute

a transformation chain only if this later is available in the engine language. Our transformation chain is made of two transformations. The first one (T0) transforms the TrML profile i.e. a UML model to a TrML model. The second transformation (T1) has for source a TrML model and for target an ATL model. However, reasoning is the same with one or more transformations in the chain. We nevertheless use two transformations in the following in order to be able to write transformation with the TrML profile and not with the TrML metamodel. To bootstrap the new language, T0 and T1 must be specified in the targeted language i.e. ATL. The only solution, at the very first time, is to write them by hand. This leads to T0_UML2TrML_v0.atl and T1_TrML2atl_v0.atl. We use the notation Tn_source2target_vx.language where *Tn* is the transformation name (T0 or T1), *source2target* represents the source and the target metamodels, *vx* is a version number and language is the language used to write and execute the transformation rule.

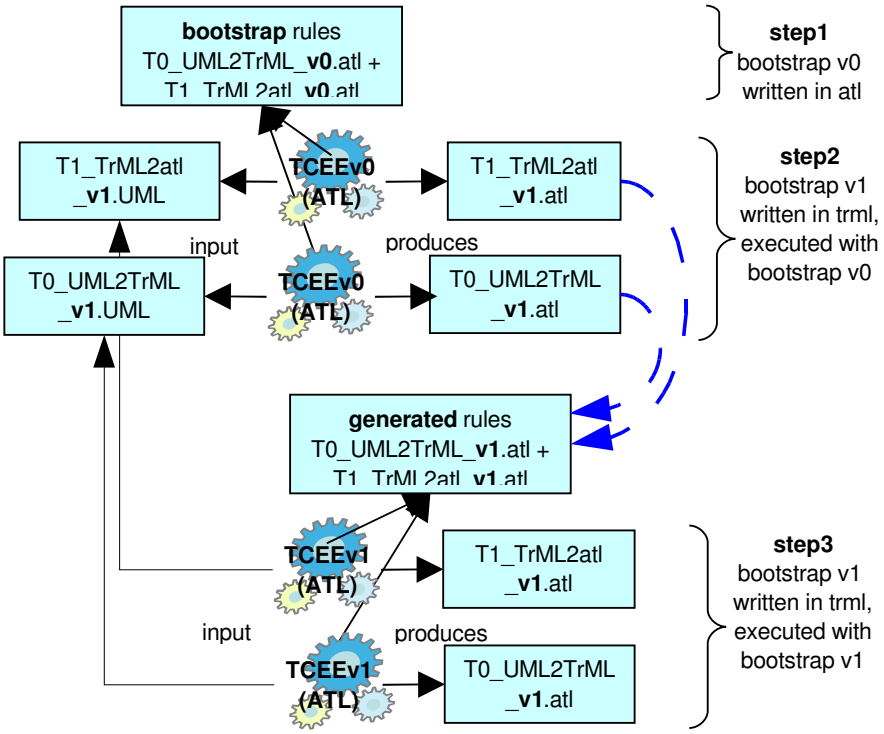

Figure 9: Bootstrapping the execution engine

The bootstrap is now complete. A transformation chain execution engine (TCEEv0) has been defined (Figure 9). Transformations written with the TrML profile can be transformed to models in ATL, and executed on the ATL engine. The second step of the bootstrap is to write T0 and T1 with the TrML profile, i.e. in UML. This corresponds to specify T0_UML2TrML_v1.UML and T1_TrML2atl_v1.UML. Thanks to the transformation chain execution engine previously defined (TCEEv0), these transformation models can be transformed in ATL and lead to new transformations executable on ATL. This leads to T0_UML2TrML_v1.atl and T1_TrML2atl_v1.atl that constitute a new transformation chain execution engine (TCEEv1). T1_TrML2atl_v0.atl and T1_TrML2atl_v1.atl may be different. Indeed, on the one hand, there are different ways to write code. On the other hand, T1_TrML2atl_v0.atl only contains the rules necessary to execute a TrML model in ATL. The complete transformation from TrML to ATL is not useful. The bootstrap mechanism allows to use transformation written with the new language to add functionalities. Nevertheless, the transformation at the model level (i.e. for example UML2RDBMS) written in TrML or in ATL leads to the same result. Thus, if we try to execute T1_TrML2atl_v1.UML no more with TCEEv0 but with TCEEv1, we obtain the same result i.e.

T1_TrML2atl_v1.atl. The transformation chain may now be improved by modifying its expression in UML, and then generating its ATL counterpart.

If we now want to be able to run the transformation chain on another engine (i.e. QVT), we should once again write the chain to transform the new language to the new targeted engine. But we are now able to write this chain using the new language (i.e. TrML profile) and to execute it with the first engine (i.e. ATL). We obtain the new chain that can be executed on the new engine (i.e. QVT). Thus, for example, when a QVT engine will be available, in order to add TrML on its top, it will be necessary only to write the transformation from TrML to QVT with the TrML profile T1_TrML2QVT_v0.UML. Indeed, a transformation chain execution engine (e.g. TCEEv1) allows executing such a transformation. This leads to T1_TrML2QVT_v1.qvt that is executable on a QVT engine. T0 doesn't need to be rewritten, it just need to be transformed to QVT with T1_TrML2QVT_v0.UML. We now have a complete transformation chain execution engine running on top of QVT.

Thus, not only TrML can be useful to document ATL zoo transformations, but can also serve as a starting point to transform them into other language as QVT. The code of the T1_TrML2atl_v0.atl is currently under development. It will be available as soon it will be stabilized.

3 Related works

Constructs and mechanisms provided by TrML depend on a strong relationships with QVT-like languages that influenced our work even if we claim having a practical user viewpoint, and having worked to product empirical research results. Presenting transformations in a graphical style naturally brings TrML towards a position very close to works on Graph Transformations [9, 5, 2, 7, 4].

Relational QVT standardizes transformation languages to help description of transformations in a concrete textual syntax. QVT specification proposes a graphical notation, but no leading editor implements it yet. Coming from Model Driven Engineering community, we find it more natural to use a graphical notation to design transformation. So, converging towards QVT graphical notation, TrML has some specificities, particularly concerning the control flow of the rules. TrML encourages coding the control within the rules during their design. Similarly, QVT proposes when and where clauses, that forces to explicitly define mutual dependencies of each rule. Thus, QVT eventually reduces reusability of the rule since the application context is hard-coded within the rule. Some other concrete syntaxes close to QVT, like ATL [6], prefer to not explicitly define the control and let it in charge of their virtual machine, thanks to a mechanism close to TrML smart reference one. Other dedicated specific transformation languages like Sitra [1] or RubyTL [3] propose constructs like bind name. Generally, model manipulation languages like EMF or Kermeta [8] allow to code transformations. But the resulting code is often hardly understandable for an external person and the intention of the transformation is often hidden by implementation details. TrML adds a graphical layer to these contributions and then increases design quality, and legibility of the transformation.

In the same objective, graphical notations coming from graph theory provide a complete support to model transformation by gathering features like specification, design, execution, validation and maintenance of transformations. Most of them have taken their benefits from graph theory and particularly graph optimization and consistency checking. In its first version TrML is focused on usability, and don' t take care about efficiency. Rule control flow is mostly external in these graphical notations. Some use state machine diagrams or activity diagrams to control execution and dependencies of the rules. Some others like [9] are close to TrML concerning the UML-like notation. But again, the philosophy of TrML is to reach a unification of graphical language that can be operationalized on any transformation environment depending of the choice of the user. Proposing a graphical layer, TrML is close to graph community contributions in Model Driven Engineering domain. They suggest raising the abstraction level from textual programming languages to visual modeling languages. However, even if the graph theory is powerful and is capable of transformation optimization, its use

is often too complicate for designers. TrML thus provides a graphical notation close to what the designers know and use (i.e. UML) without applying graph transformation techniques for model transformation. It lets user free to choose any implementation. Our approach allows to use one of the existing transformation language to code the transformations and does not force designers to use graph theory.

Globally, our experience shows that choosing a particular language or method to design rules before knowing the real nature of the transformation seems not to be a perennial approach. In fact, the early choice of a language or tool may reduce flexibility and restrain the transformation design.

4 Conclusion

This paper shows how it is possible to build and execute a new transformation language on top of an existing one.

The proposed new language shows that it is possible to design graphically transformations with a standard UML profile. This transformation design can still be independent from the targeted transformation engine. It facilitates exchange and communication, giving an abstraction of the manner the transformation is released.

TrML provides a prototype of graphical notation that tries to unify visual transformation design based on simple design elements close to UML classical representation. Moreover, TrML gives the possibility to use main UML modeler tools, by drawing transformation with the profile form of TrML.

Different experiments in our design team or in workshops using a graphical notation have shown that it provides more visibility on the purpose of the transformation than textual languages and immediately initiates exchanges. The related works part shows that different existing languages propose various attractive facilities to help coding a transformation, like OCL Helpers, where and when directives, bind names... TrML tries to gather the minimum and best of them needed by designers to describe transformations. Large engineering projects take advantage to be graphically designed in an abstract form, and then be precisely described in different formalisms according to specific concerns. TrML relies on this observation and applies the same principle to transformation design. It has been assessed on systems on chips product chains using model transformation to cope with different concerns of this type of system.

The paper also shows that the new language can be executed on top of existing transformation engines. This requires a manual bootstrapping on top of an existing engine, using the targetted engine language. Execution on top of other engines can now be implemented by writing a transformation in the new language itself. Furthermore, the new language can now be improved or rewritten by using itself.

References

[1] D. H. Akehurst, B. Bordbar, M. Evans, W. G. Howells, and K. D. McDonald-Maier. SiTra: Simple transformations in java. In *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences)*, Genova, Italy, October 2006.

[2] A. Balogh and D. Varrò. Advanced model transformation language constructs in the viatra2 framework. In *ACM Symposium on Applied Computing, Model Transformation Track*, 2006.

[3] J. Sanchez Cuadrado, J. Garcia Molina, and M. Menarguez Tortosa. Rubytl: A practical, extensible transformation language. In Lecture Notes in Computer Science, editor, *European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA*, 2006.

[4] J. de Lara and H. Vangheluwe. Atom3: A tool for multi-formalism modelling and meta-modelling. In Lecture Notes in Computer Science, editor, *FASE/ETAPS'02*, volume 2306, pages 174–188, 2002.

[5] L. Grunske, L. Geiger, and M. Lawley. A graphical specification of model transformations with triple graph grammar. In Lecture Notes in Computer Science, editor, *First European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA*, volume 3748, pages 284–298, November 2005.

[6] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM*, Montego Bay, Jamaica, October 2005.

[7] Lengyel L, Levendovszky T, Mezei G, Forstner B, and Charaf H. Metamodel-based model transformation with aspect-oriented constraints. In *GraMoT'05 - International Workshop on Graph and Model Transformation,*, 2005.

[8] Pierre-Alain Muller, Franck Fleurey, Zoé Drey Didier Vojtisek, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, October 2005.

[9] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE*, volume 3062, pages 446–453, 2003.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399