Tool support for MOLA

Audris Kalnins [1], Edgars Celms [2,3], Agris Sostaks [4]

IMCS
University of Latvia
Riga, Latvia

Abstract

The paper describes the MOLA Tool, which supports the model transformation language MOLA. MOLA Tool consists of two parts: MOLA definition environment and MOLA execution environment. MOLA definition environment is based on the GMF (Generic Modeling Framework) and contains graphical editors for metamodels and MOLA diagrams, as well as the MOLA compiler. The main component of MOLA execution environment is a MOLA virtual machine, which performs model transformations, using an SQL database as a repository. The execution environment may be used as a plug-in for Eclipse based modeling tools (e.g., IBM Rational RSA). The current status of the tool is truly academic.

Key words: Model transformations, MDD, MOLA, MOLA tool.

1 Introduction

Practical use of Model Driven Development (MDD) for building systems is impossible without appropriate tools. Principles of MDA and MDD are known for quite a time and several model transformation languages, including the emerging OMG standard (QVT-Merge) [15] have got certain publicity. However, there are very few truly MDD tools available. At the time of writing, the available commercial tools supporting MDD (OptimalJ[14], ArcStyler[2], Objecteering[13] and other) do it well for specific kinds of PSM (frequently, J2EE) and specific design methodologies, but modifying the used default model transformations is as hard as extending traditional modeling tools - in most cases conventional OOP languages are used to define transformations. On the other hand, the experimental model transformation tools - ATL[3],

[1] Email: Audris.Kalnins@mii.lu.lv
[2] Email: Edgars.Celms@mii.lu.lv
[3] supported partially by ESF
[4] Email: agree@os.lv

MTF[12], Tefkat[16], etc. which are mainly Eclipse EMF based and use various (mainly textual) transformation languages, are not well linked with the model providers - the modeling tools.

In this paper the academic MOLA tool, which is being developed at the University of Latvia and supports the graphical model transformation language MOLA [7], is described. The goal of the design has been to have a simple implementation, which nevertheless would be practically usable in the MDD context. The structure and main principles of the tool are described, and also its links with modeling tools. In a more detailed way, it is shown how MOLA execution environment can be linked to Eclipse EMF based modeling tools. This is illustrated by a case study - an application of MDD principles to IS design based on Hibernate framework.

2 Brief Description of MOLA

The MOLA tool is based on the MOLA model transformation language, developed at the University of Latvia, IMCS [7,8,9,10]. MOLA is a graphical procedural transformation language. Its main distinguishing features are advanced graphical pattern definitions and control structures taken from the traditional structural programming. To facilitate the understanding, we briefly remind the main concepts of MOLA.

Like most of the model transformation languages, MOLA is based on source and target metamodels, which describe the source and target models respectively. The used metamodeling language is EMOF [11](with some slight restrictions). In MOLA source and target metamodels are combined in one class diagram, but packages may be used for structuring. The source and target may coincide. Special **mapping associations** linking the corresponding classes in source and target metamodels may be added to the metamodel. Their role is similar to relations in other transformation languages - for structuring the transformation and documenting the transformation traceability.

The transformation itself is defined by one or more **MOLA diagrams** (see examples in Fig. 6 and 7). A MOLA diagram is a sequence of graphical statements, linked by arrows. The most used statement in a MOLA diagram is the **FOREACH loop** - a bold-lined rectangle. A loop has a **loop head** (a grey rounded rectangle), which contains the **loop variable** (bolded element) - a class, instances of which the loop has to iterate through. In addition, the loop head contains a **pattern**, which specifies namely which of the instances qualify for the given loop. A pattern is a metamodel fragment, but in instance notation - `element_name:class_name`, therefore classes may be repeated. Links just correspond to metamodel associations. A pattern element may contain an attribute-based constraint - an expression in OCL subset. The semantics of loop is quite natural - the loop must be executed for all instances of the loop variable for which there exist instances of other pattern elements satisfying their constraints and linked by the specified links (pure existence

semantics). Loops may be nested, the instance of the loop variable (and other elements) matched in the parent loop may be referenced in the nested loop by the reference notation - the element name prefixed by @ character.

Another kind of graphical statements is the **rule** (a grey rounded rectangle too), which also contains a pattern but without loop variable. A rule typically contains actions - element or association building (red dotted lines) and deletion (dashed lines). A rule is executed once in its control path (if the pattern matches) or not at all - thus it plays the role of an if-statement too. A loop head may also contain actions. MOLA subprograms are invoked by the call statement (possibly with parameters).

One year experience of using MOLA (mainly in academic environment - from undergraduate to PhD students) has confirmed its ease of learning and high readability of defined transformations - especially when compared to the current QVT-Merge proposal [15].

Actually, quite a few graphical model transformation languages are now in use - besides the graphical form of QVT-Merge, Fujaba Story diagrams (SDM) [6] and the GME-based GReAT notation [1] is used. The pattern definition facilities are approximately of the same strength in all these approaches, including MOLA. There are differences in defining the rule control structure, the Fujaba approach is the closest one to MOLA, but is less structured, while GReAT is more based on data flows. Actually we don't mention here the graph transformation languages, which have slightly different goals. A more comprehensive comparison of MOLA to other languages has been already given in [7].

3 The Architecture of MOLA Tool

The current version of MOLA tool has been developed with mainly academic goals - to test the MOLA usability, teach the use of MDD for software system development and perform some real life experiments. This has influenced some of the design requirements, though with easy usability as one of the goals and sufficient efficiency the tool has confirmed its potential as an industrial tool too.

Like most of the model transformation tools, the MOLA tool has two parts - the **Transformation Definition Environment (TDE)** and the **Transformation Execution Environment (TEE)**. Both these environments have a common repository for storing the transformation, metamodels and models (in the runtime format). Fig. 1 shows the general architecture of the tool.

The definition environment is related to the metamodel level - M2 in the MOF classification. Its intended users are methodology experts who provide the metamodels and define the transformations for development steps which can be automated. Since MOLA is a graphical language, TDE is a set of graphical editors, built on the basis of GMF [4] - a generic metamodel based modeling framework, developed by University of Latvia, IMCS together with

the Exigen company.

The execution environment (related to M1 level) is intended for use by system developers, who according to the selected MDD methodology perform the automated development steps and obtain the relevant target models. Currently two forms of TEE are available. The form closer to an industrial use is an Eclipse plug-in, which can be used as a transformation plug-in for UML 2.0 modeling tools, including the commercial IBM Rational tool RSA. This use is described in more details in section 5 and demonstrated on a case study. Another form is a more experimental one. It is based on GMF as a generic modeling environment and is intended for various domain specific modeling and design notations. It is described more closely in section 6.

Fig. 1 shows both the components of the MOLA tool (rounded rectangles) and the used data objects (rectangles). Besides the traditional class diagram notation, arrows represent the possible data flows. Data objects in MOLA runtime repository are annotated as tables because it is SQL based.

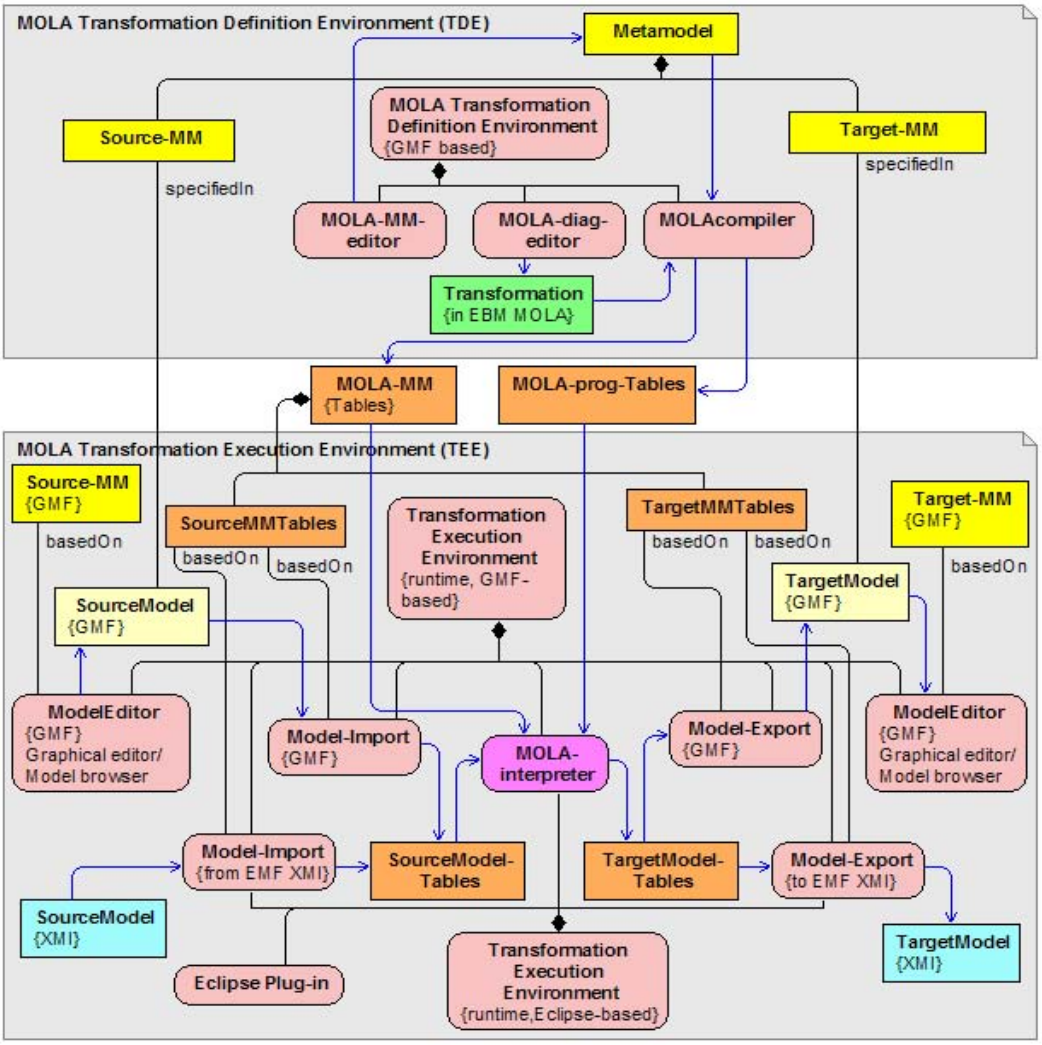

Figure 1. MOLA Tool environment architecture.

Now some more comments on the MOLA TDE. It contains graphical editors for class diagrams (EMOF level) and MOLA diagrams. Both the source and target metamodels are shown in the same class diagram, together with possible mapping associations. A transformation is typically described by several MOLA diagrams, one of which is the main. Since the graphical editors are implemented on the basis of GMF, they have professional diagramming quality, including automatic layout of elements. In addition to editors, TDE contains the MOLA compiler which performs the syntax check and converts both the combined metamodel and MOLA diagrams from the GMF repository format to the MOLA runtime repository format. Fig. 2 shows a screenshot of MOLA TDE, with both metamodel and MOLA diagram editors open.

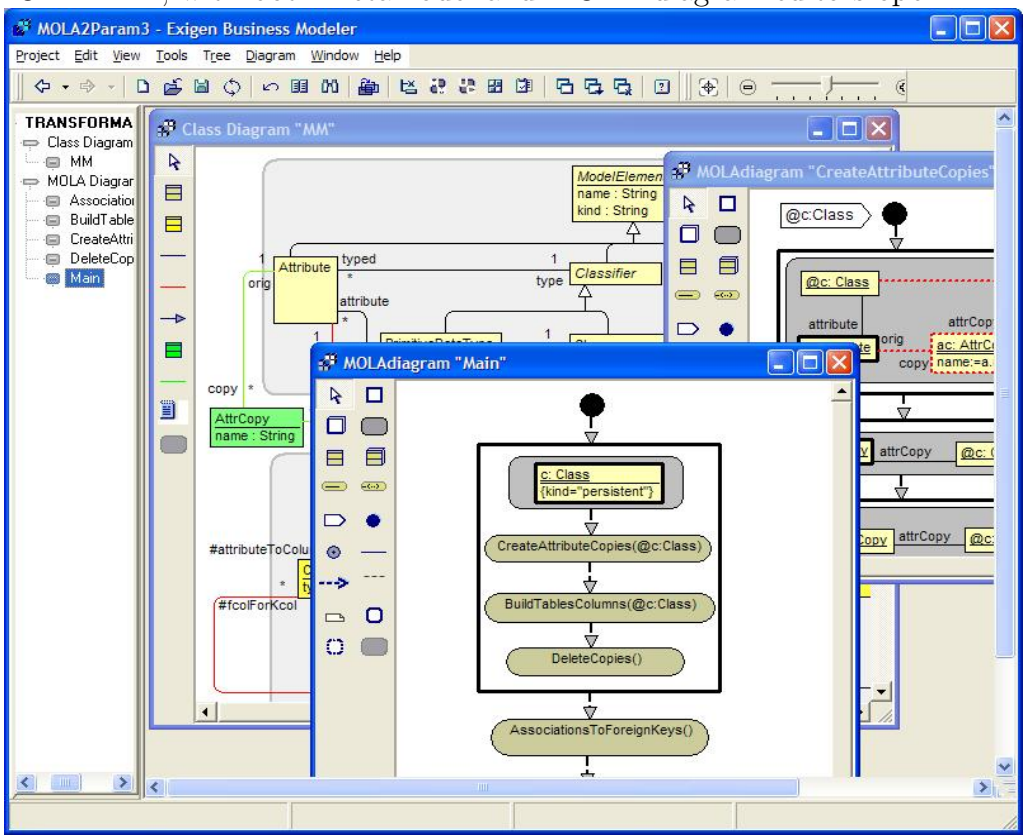

Figure 2. Screenshot of the MOLA TDE.

4 MOLA Virtual Machine and Repository

The core of the MOLA TEE is MOLA Virtual Machine (**VM**) - an interpreter performing the model transformation. Certainly, it is closely linked to the MOLA repository, whose main function is to ensure the efficiency of MOLA VM. The most crucial factor in implementing MOLA VM is the implementation of pattern matching - the most "expensive" part in any transformation implementation.

Model transformation tools [3,12,16] typically are implemented on meta-

model based repositories such as Eclipse EMF. Such an implementation typically uses low level repository operations for pattern matching and is more compiler than interpreter. Authors of this paper have shown [10] that a very efficient MOLA implementation is possible this way. However, for this academic implementation of MOLA another goal was set - the implementation must be as simple as possible, but still usable on examples of reasonable size, e.g., to transform a model containing several hundred classes.

Therefore another solution was adopted - to use an SQL database as a repository. The key rationale for this solution has been that a complete MOLA pattern match operation can be implemented by one SQL Select statement. In addition, this Select statement can be generated very easily from the MOLA pattern definition also stored in SQL tables in an appropriate way by the MOLA compiler. Thus MOLA VM would be quite close to a pure interpreter and therefore simple. The implementation of other MOLA elements is not so sensitive to repository format because it is a fairly classic implementation of traditional control structures. Namely these principles are used for MOLA VM implementation in this version of MOLA tool.

The only problem which remained to be solved was efficiency. The query generated from a MOLA pattern is quite untypical for standard SQL databases - it is a "self-join" of two tables (representing class and association instances in the model) as many times as there are elements and links in the pattern. Not all database engines occurred to process such joins satisfactorily. Since the desire was to build the MOLA tool as open source based as possible, the first candidate was MySQL. However, it occurred that for large patterns the performance was not satisfactory - the query optimization itself used by the MySQL engine was too lengthy for this type of queries. Another candidate was the free version Microsoft SQL - the MSDE engine. It performed quite efficiently for patterns occurring in reasonable MOLA programs and quite large example models. This way the stated goals for both simple and efficient MOLA VM were reached.

5 MOLA Transformation Execution Environment as an Eclipse plug-in

The MOLA TEE besides the MOLA VM must contain components for fetching the source models to be transformed and passing the transformation results. In a typical MDD scenario there must be also a modeling environment where the source models are prepared and the obtained target models are processed further. In the approach where MOLA transformations are used as plug-ins for Eclipse EMF namely this scenario is assumed. The environment was selected due its popularity as a model transformation testbed and because there is a publicly available UML 2.0 metamodel [17] for this environment. The MOLA TEE in this case, in addition to MOLA VM, contains XMI import component, XMI export component and a simple Eclipse plug-in (see the lower

layer of TEE in Fig. 1). The XMI import component currently supports a reasonable subset of UML 2.0 metamodel (in its EMF version). The XMI export component also supports a subset of UML 2.0, but in addition some other metamodels available in EMF are supported (e.g., the SQL database definition metamodel). The plug-in currently is very simple - it is used just to activate the TEE and select the source and target XMI location and the required MOLA transformation. The source model must be exported by the Eclipse modeling tool export facility and the generated target model imported by the import facility.

Currently this schema has been tested with the only professional Eclipse EMF based modeling tool truly supporting UML 2.0 - IBM Rational RSA. As soon as more Eclipse based modeling tools support the UML 2.0 metamodel, the same plug-in would be applicable to them. At the time of writing, there is no true transformation plug-in for RSA (the embedded RSA transformation extension facilities require coding in Java), so the developed plug-in could present also some practical interest. Certainly, a more user friendly solution would be to acquire the relevant source model directly via Eclipse API and pass the result this way too, but this solution is much more complicated and more tied up to a specific modeling tool.

The proposed solution seems to be practically usable for various MDD style development scenarios. Section 7 contains one such case study - a scenario where the Hibernate persistence framework is used. Since MOLA is well suited for model-to-model transformations, but not so well for model-to-code, the built-in code generation facilities of RSA (or other modeling tool) are used for this purpose.

6 Standalone MOLA Transformation Execution Environment

Another possibility to have a usable transformation execution environment is to tie the MOLA VM up to a generic modeling environment where an arbitrary graphical modeling notation can be supported. Since the GMF environment [4] is such one, another MOLA TEE has been based on it. Truly speaking, Eclipse+EMF+GEF [5] is also such an environment, but the development there requires much more effort. The MOLA TEE version based on GMF is meant for various experiments in applying MDD and model transformations to domain-specific notations, including non-UML ones. The top layer of TEE in Fig. 1 shows the corresponding components. In GMF it is possible to define the graphical presentation of a domain model as a sort of transformation (though not very universal, see more in [4]), therefore for many modeling notations usable graphical editors can be defined without proper programming at all. In any case, Eclipse EMF style model browsers/editors, but more flexible ones, can be built very easily with GMF. Universal metamodel-controlled export and import components from/to GMF repository have been built. This

task has not been so hard since the GMF repository is functionally close to EMOF. The relevant MOLA transformation can be invoked directly from this environment (MOLA VM is used as a GMF plug-in).

Several such experiments have been performed. In one case, a UML activity diagram profile (a complicated one and represented graphically) meant for defining workflows in UML was implemented in GMF, and a transformation to a specific workflow notation was defined in MOLA. GMF has a special feature of generating readable diagrams automatically, therefore in many cases the transformed target model can be automatically presented as a diagram.

Another GMF based experiment has been in converting a special profile of class diagrams to OWL notation for ontology definitions.

7 Case Study: Use of MOLA Tool for Building an IS within Hibernate Framework

In this section we show how the Eclipse plug-in form of MOLA tool could be used for MDD style development of information systems in Java within the Hibernate persistence framework. This framework provides a "classical" object-relational mapping between Java classes and database tables, permitting a developer to access instances of such classes (actually stored in a database) as if they were true Java objects. The modeling tool where the models are built is assumed to be IBM Rational RSA. Just one nontrivial step in the methodology is illustrated. We assume that a PIM - an IS domain model in the form of a standard UML 2.0 class diagram has been built (see a small example in Fig. 3).

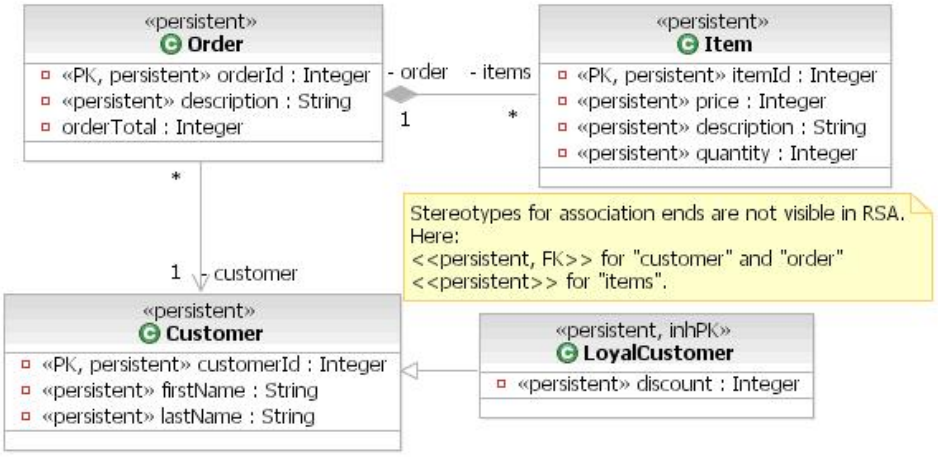

Figure 3. UML class diagram with stereotypes in RSA (PIM model).

Some of the classes must be persistent - stored in a relational database as tables. The standard Hibernate mapping is assumed for these classes, which requires "standard" Java getters and setters to be added for the persistent attributes and association ends of a class, with other attributes and operations unmodified. In addition, for each such mapping the Hibernate mapping de-

scriptor (an XML file) must be built. Thus the task is to build a PSM model consisting of three parts - the augmented UML classes, database schema definition and Hibernate mapping descriptors.

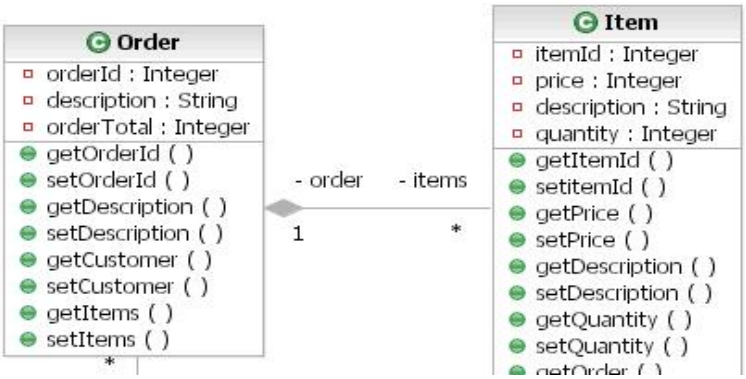

Figure 4. Part of class diagram for Hibernate framework in RSA(PSM model).

In order to specify adequately the logical design decisions at the PIM level, a custom profile (`HibernateProfile`) is required. This profile should contain the stereotypes `persistent` (both for classes and properties, representing either attributes or association ends), `PK` - for attributes (properties), `FK` - for association ends (properties) and `inhPK` - for defining Hibernate-style storing of persistent subclasses. If these stereotypes are appropriately applied to the PIM model, then the three-part PSM can be generated automatically by a MOLA transformation - for each persistent class a table will be defined (containing persistent attributes and associations), getters/setters will be added to the class and the Hibernate descriptor will be defined. Fig. 3 shows these stereotypes applied (RSA does not visualize stereotypes for association ends). All classes there are assumed to be persistent, but not all attributes. Classes in a PIM normally should contain also business operations, we don't show them for brevity. We assume also that primary keys consist of one column (Hibernate uses a complicated mapping for complex keys).

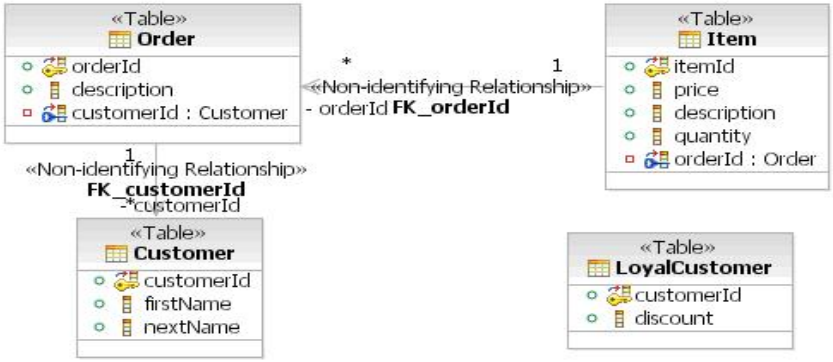

Figure 5. Data model visualization in RSA (PSM model).

Fig. 4 shows the first component of the result - the updated class diagram (fragment). Getters and setters are added where appropriate, but custom stereotypes are removed - RSA does not use them for code generation.

Fig. 5 shows the second component of the result - the database schema.

The model is built according to the EMF SQL metamodel, but RSA data model visualization feature is used to show the schema as a diagram.

Finally, Fig. 6 and 7 show the part of the MOLA transformation - the main program and SQL table building.

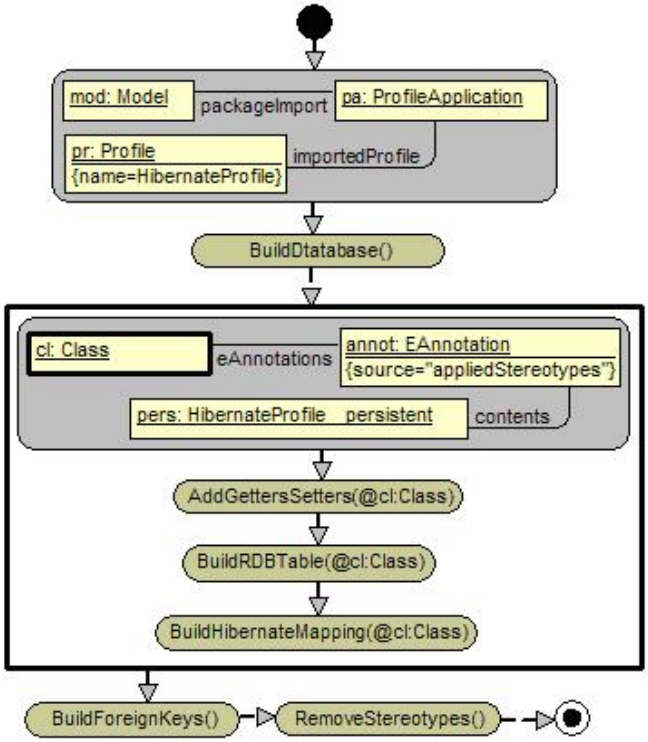

Figure 6. MOLA transformation program (main).

The metamodels are not shown due to lack of space. The source metamodel is the standard UML 2.0 metamodel. However, in EMF a special coding (not the OMG standard, but eCore defined via the `EAnnotation` metaclass) is used for applied stereotypes. Namely this coding is used in transformations - in UML 2.0 the applied stereotypes show up as instances in the model, therefore model transformations must treat them as instances of special temporary metaclasses (note the MOLA pattern for finding a persistent class in the FOREACH loop of Fig. 6). The `AddGettersSetters` transformation (not shown here) uses the same UML metamodel as a target - it is an update transformation, which simply attaches new `Operation` instances to existing `Class` instances. The `BuildRDBTable` program (Fig. 7) builds a table for the class and then performs a loop, which builds a column (including its type and key constraints) for each persistent property. The target metamodel for this program is the SQL metamodel in EMF, but for `BuildHibernateMapping` - the metamodel obtained from the Hibernate XML schema definition. Actually in MOLA all these metamodels appear as a common class diagram, but packages are used to separate them. The same packages are used to guide the MOLA tool XMI exporter component - in this case several separate XMI files must be generated, but for Hibernate mapping a non-XMI XML coding is required.

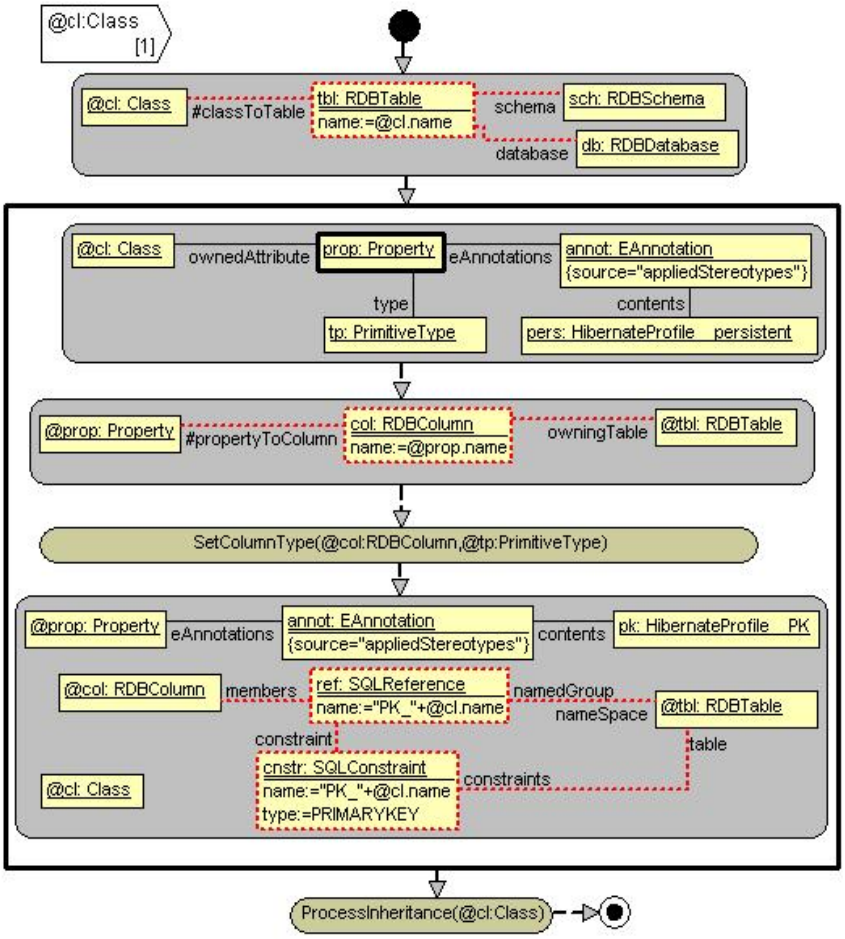

Figure 7. MOLA transformation program (BuildRDBTable).

8 Conclusions

The structure and some use cases of the experimental academic MOLA tool have been described in the paper. The existing experience of using the tool has shown that the adopted solutions are appropriate and MOLA transformations fit in well both in the traditional UML based MDD style development and in domain specific modeling. Certainly, the practical tool usability has to be improved, especially the links with the modeling tools. One more issue to be solved is the "round-tripping", because in MDD setting the target models are also sometimes updated manually. MOLA transformations have no "native reversibility", but it is clear that for typical MDD tasks reverse transformations are easy to build, using either mapping associations (in standalone environment) or special annotations (in EMF environment). Yet another task is to build a MOLA transformation library for typical MDD use cases.

References

[1] Agrawal A., G. Karsai, F. Shi. "Graph Transformations on Domain-Specific Models". Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, 2003

[2] ArcStyler. URL: http://www.interactive-objects.com/

[3] ATL. URL: http://www.sciences.univ-nantes.fr/lina/atl/

[4] Celms E., A. Kalnins, L. Lace. "Diagram definition facilities based on metamodel mappings". Proceedings of the 18th International Conference, OOPSLA'2003 (Workshop on Domain-Specific Modeling), Anaheim, California, USA, October 2003, pp. 23-32.

[5] Eclipse GEF. URL: http://www.eclipse.org/gef/

[6] Fujaba User Documentation. URL: http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf

[7] Kalnins A., J. Barzdins, E. Celms. "Model Transformation Language MOLA". Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004), Linkoeping, Sweden, June 10-11, 2004. pp.14-28.

[8] Kalnins A., J. Barzdins, E. Celms. "Basics of Model Transformation Language MOLA". ECOOP 2004 (Workshop on Model Transformation and execution in the context of MDA), Oslo, Norway, June 14-18, 2004. URL: http://heim.ifi.uio.no/ janoa/wmdd2004/papers/

[9] Kalnins A., J. Barzdins, E. Celms. "MOLA Language: Methodology Sketch". Proceedings of EWMDA-2, Canterbury, England, 2004. pp.194-203.

[10] Kalnins A., J. Barzdins, E. Celms. "Efficiency Problems in MOLA Implementation". 19th International Conference, OOPSLA'2004 (Workshop "Best Practices for Model-Driven Software Development"), Vancouver, Canada, October 2004. URL: http://www.softmetaware.com/oopsla2004/mdsd-workshop.html

[11] MOF 2.0 Core Final Adopted Specification. URL: http://www.omg.org/docs/ptc/03-10-04.pdf

[12] MTF. URL: http://www.alphaworks.ibm.com/tech/mtf

[13] Objecteering. URL: http://www.objecteering.com/

[14] OptimalJ. URL: http://www.compuware.com/products/optimalj/

[15] QVT-Merge. URL: http://www.omg.org/docs/ad/05-03-02.pdf

[16] Tefkat. URL: http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/

[17] UML 2.0 Eclipse EMF. URL: http://www.eclipse.org/uml2/