Horizontal Transformations of Visual Models in MetaLanguage System

Alexander O. Sukhov
Department of Software and Computing Systems Mathematical Support
Perm State University
Perm, Russian Federation
E-mail: Sukhov.psu@gmail.com

Scientific Advisor:
Lyudmila N. Lyadova
Department of Business Informatics
National Research University Higher School of Economics
Perm, Russian Federation
E-mail: LNLyadova@gmail.com

***Abstract*. Different specialists are involved in software development at once: databases designers, business analysts, user interface designers, programmers, testers, etc. It leads to creation and usage in systems designing of various models fulfilled from the different points of view, with different levels of details, which use different modeling languages for the description. Thus there is a necessity of models transformation as between different levels of hierarchy, and within the same level between different modeling languages for creation of united model of system and exporting of models to external systems. The MetaLanguage system is intended to visual domain-specific languages creation. The approaches to development of a model transformation component of MetaLanguage system are considered. This component allows to fulfill vertical and horizontal model transformations of "model-text" and "model-model" types. These transformations are based on graph grammars described by production rules. Each rule contains the left- and right-hand sides. The algorithm of the left-hand side search in the source model and the algorithms of execution of a right-hand side of a rule are described. Transformations definitions for models in ERD notation are presented as example.**

Keywords: model-based approache; visual model; domain-specific language; horizontal model transformation; language workbench.

I. INTRODUCTION

In industrial production we often come to the fact that the studying and creation of an object is done by constructing its model. Since development of computer systems the idea of creation and usage of models has come to computer science.

Model is an abstract description of system (object, process) which contains characteristics and features of its functioning which are important from the viewpoint of modeling purposes. *Metamodel* is a language used for models development. For metamodels description the *meta-metamodels* (*metalanguages*) are used. *Modeling* is the process of creation and studying of models.

Today the majority supposes that visual models are used only at the early stages of software development, for creation of certain "sketch" of system or transfer of high-level ideas of designing, i.e. it is supposed that models play a secondary role, and are primarily used only for documentation. However there are approaches to system engineering in which the basic elements are the visual models and their transformations – model-based approaches.

The model-based approaches are capable at information system creation to unite efforts of developers and domain experts. These approaches make the system more flexible, since for its change there is no necessity of modification of source code "by hand", it is enough to modify a visual model, and with this task even nonprofessional programmers can cope.

For model-based approaches implementation it is necessary to use toolkit which will be convenient to various participants of system development process. The general-purpose modeling languages, such as UML, are not able to cope with this task, because they have some disadvantages:

- Diagrams are complicated for understanding not only for experts, who take part in system engineering, but in some cases even for professional developers.
- Object-oriented diagrams can not adequately represent domain concepts, since work is being done in terms of "class", "association", "aggregation", etc., rather than in domain terms.

That is why at implementation of model-based approaches the domain-specific modeling languages (DSMLs, DSLs), created to work in specific domains, are increasingly used. Domain-specific languages are more expressive, simple on applying and easy to understand for different categories of users as they operate with domain terms. Therefore now a large number of DSLs is developed for using in different domains [1-3].

Despite all DSLs advantages they have one big disadvantage – complexity of the designing. If general purpose languages allow creating programs irrespectively to domain, in case of DSLs for each domain, and in some cases for each task it is necessary to create the domain-specific language. Another shortcoming of domain-specific language is that it is necessary to create convenient graphical editors to work with it.

This paper is supported by Russian Foundation for Basic Research (Grant 12-07-00763)

The *language workbench* or *DSM-platform* is the instrumental software intended to support development and maintenance of DSLs. Usage at DSLs creation a language workbench considerably simplifies the process of their designing. The *MetaLanguage* [4] system is a language workbench for creating visual dynamic adaptable domain-specific modeling languages. This system allows fulfilling multilevel and multi-language modeling of domain.

The different categories of users work at various stages of system life cycle. At the stage of system creation the leading role is played by professional developers with participation of experts, specialists in the appropriate domain, and at the operation stage – by experts, specialists and end-users, as they detect all system shortcomings and mistakes in its implementation. To attract experts and specialists to the process of system adjustment of the ever-changing operating conditions and user requirements it is necessary to provide them with the convenient language, which is operates with customary terms. Using this language they could make all necessary modifications of information system.

On the other hand, several specialists are involved in software development at once: databases designers, business analysts, user interface designers, programmers, testers, etc. Each of these specialists uses their own information about the system and this information may describe the same objects, but from the different points of view and with various modeling languages (see fig. 1).

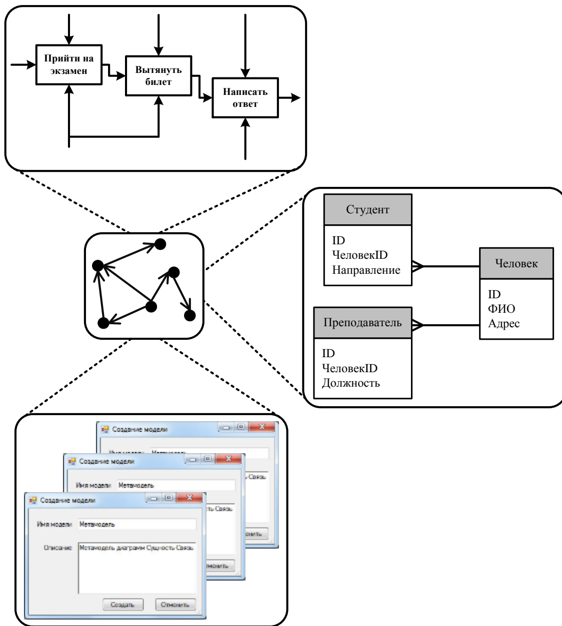


Fig. 1. Consideration of system objects from different points of view

Thus the software development process includes various types of activity in which different categories of users participate. It leads to creation and usage in systems designing of *various models* fulfilled from the different points of view, with different levels of details, which use for the description different modeling languages (see fig. 2).

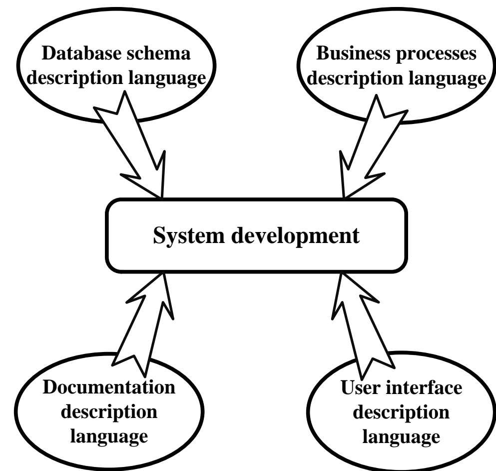


Fig. 2. The usage of different languages for software development

So there is a necessity of models transformation as between different levels of hierarchy, and within the same level between different modeling languages, for creation of united model of system and exporting of models to external systems (see fig. 3).

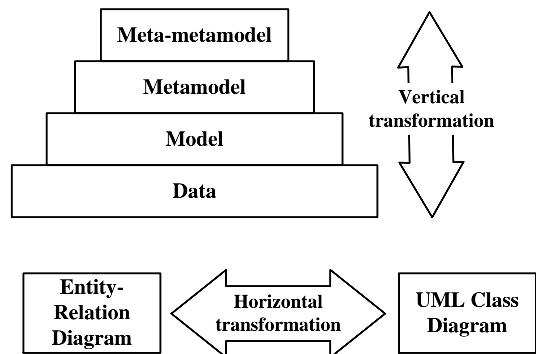


Fig. 3. Vertical and horizontal model transformations

In addition, there is an unresolved problem of models exporting from one information system to another (for example, business processes described in one system can not be executed in another, that these systems use various notations for business processes description).

Usage of domain-specific languages and tools for their creation also affects a transformation problem as there is a need of export of the created by the user models to external systems which, as a rule, use one of the standard modeling languages that is different from developed DSL. That is why one of the main components of the MetaLanguage system is the *Transformer*. This component uses graph grammars for transformations describing. Implementation of graph grammars in the MetaLanguage system is defined by assignment of this language workbench.

II. Basic Concepts

The basic concept of transformation definition is a *production rule* which looks like $p: L \rightarrow R$, where p is a rule *name*, L is a *left-hand side* of the rule, also called the *pattern*, and R is a *right-hand side* of the rule, which is called the

replacement graph. Rules are applied to the starting graph named the *host-graph*.

Let's suppose that four labeled graphs G, H, L, R are given, and graph L is a subgraph of graph G. Applying of the rule $p: L \rightarrow R$ to the starting graph G is called the replacement in graph G of subgraph L on graph R, which is a subgraph of graph H. The graph H is the result of this replacement.

Graph grammar is a pair $GG = (P, G_0)$, where P is a set of production rules, G_0 is a grammar starting graph.

Graph transformation is a sequenced applying to the starting labeled graph G_0 of finite set of rules

$$P = (p_1, p_2 \cdots p_n): G_0 \overset{p_1}{\Rightarrow} G_1 \overset{p_2}{\Rightarrow} ... \overset{p_n}{\Rightarrow} G_n .$$

List of production rules is arranged according to the chosen discipline, for example, by priorities. The transformation process is completed when the list of rules does not contain any rule which can be applied. There are also other disciplines of rules ordering, so some systems use the control mechanism to explicitly specify which rule should be applied as follows [5].

At transformations direction they can be classified as vertical and horizontal. *Vertical transformations* convert the models which belong to various hierarchy levels, for example, at mapping of the metamodel objects to domain model objects. *Horizontal transformation* is the conversion, in which the source and target models belong to one hierarchy level. An example of a horizontal transformation is a conversion of model description from one notation to another (see fig. 3).

The models are described with some modeling languages. Depending on the language on which source and target models are described, horizontal transformations can be divided into two types: endogenous and exogenous. *Endogenous transformation* is the transformation of the models which are described on the same modeling language. *Exogenous transformation* is the transformation of models which are described on various modeling languages [6].

Graph grammar often used to describe of any transformations, performed on graphs: definition of the models operational semantics [7, 8], the analysis of program systems with dynamic evolving structures [9, 10], etc. These grammars allow to describe the transformations that should occur in system at performance over it of the operations, specified in grammar.

The right-hand side of the rule may be not only a labeled graph, but the code on any programming language, and also a fragment of a visual model described in some notation. That is why the graph grammar can be used for generation syntactic correct models and for refactoring of existing models, code generation and model transformations from one modeling language to another [11].

Considering singularities and designation of MetaLanguage system, it is necessary to make the following requirements to its transformation component:

- To be obvious and easy to use for providing the opportunity of involving to transformation description not only programmers, but also experts, specialists in domains. It can be achieved through the usage of visual notation of transformations description language.
- To allow using the created transformations directly in the system, i.e. to produce the models transformation in the same user interface in which they were designed.
- To perform both horizontal and vertical transformations, and availability of possibility to fulfill the horizontal transformations from one notation to another, including a "model-text" type.
- Metamodels from the left- and right-hand sides of the rule can be described by a user created metalanguage.
- To allow specifying the transformations of entities and relations attributes and constraints imposed on metamodel elements.

III. Related Works

There are various approaches to model transformations, some of them have the formal basis, so the systems AGG, GReAT, VIATRA use graph rewriting rules to perform transformations, and others apply technologies from other areas of software engineering, for example, the technique of programming by example.

Various modifications of the algebraic approach are implemented in systems AGG, GReAT, VIATRA. In AGG [12] the left- and right-hand sides of the production rule are the typed attribute graphs, both sides of a rule should be described in one notation, i.e. this system allows to fulfill only endogenous transformations that does impossible its usage in MetaLanguage system. Besides, this tool does not allow to make transformation of a "model-text" type. However the usage as the formal basis of the algebraic approach to graph transformations allows to produce graph parsing, to verify graph models, and the extension of graphs of Java possibilities makes transformations more powerful from a functional point of view.

The GReAT system [13] is based on the algebraic approach with double-pushout, therefore for transformation description it is necessary to create the domain that contains both the left- and right-hand sides of the production rule simultaneously with instructions of what element it is necessary to add, and what to remove. This form of rule is unusual for the end-user and a bit tangled. However it provides a possibility of execution the transformation of several source metamodels at once, which is significant advantage in comparison with other approaches. For metamodels definition the GReAT uses UML and OCL, it does not allow the user to choose the language of metamodels specification or to change its description. It makes this approach unsuitable for usage in MetaLanguage.

The QVT (Query/View/Transformation) is the proposed by OMG approach to models transformation, which provides the user with declarative and imperative languages [14]. Conversion is defined at the level of metamodels which is described on MOF. The advantage of this approach is the existence of standard of its description, and also usage of

standard languages OCL and MOF at the models transformation definition process. However these advantages also have the other side. Usage of MOF as a meta-metamodeling language, does not allow the user to choose a metalanguage convenient for him, or to change description of the metalanguage which is integrated in the QVT. In addition, this approach does not allow to make the transformation of a "model-text" type, since each metamodel should be described using the MOF standard. It imposes of some restrictions on a possibility of QVT usage in the MetaLanguage system.

VIATRA [15] is a transformation language, based on rules and patterns, which combines two approaches into a single specification paradigm: the algebraic approach for models description and the abstract state machines intended for exposition of control flow. Thanks to constructions of state machines the developers significantly raised the semantics of standard languages of patterns definition and graph transformation. Besides, powerful metalanguage constructions allow to make multi-level modeling of domains.

One of shortcomings of the VIATRA is an inexpressive textual language of metamodels description. Although the developers of approach have criticized the MOF standard for the lack of a possibility of multi-level modeling, they still remain within limits of this paradigm at usage of visual language for metamodels definition. VIATRA is not intended for execution of horizontal model transformations. Its main purpose is a verification and validation of the constructed models by their transformation.

The ATL is the language, allowing to describe transformations of any source model to a specified target model [16]. Transformation is performed at the level of the metamodels. The heart of ATL is the standard language of constraints description OCL.

The disadvantage of this language is high requirements to the conversion developer. Since ATL in most cases uses only textual definition of transformation, then in addition to knowledge of source and target metamodels the developer needs to know language of transformation definition. Lack of navigation on the target model complicates the process of rules determination.

The ATL is a dialect of QVT language and therefore inherits all its shortcomings. One of the differences from the QVT is very strict restriction on created transformations: the left-hand side of the rule should contain only one element. It highly complicates the development, increasing an amount of rules in system. All it does impossible the usage of this approach in the MetaLanguage system.

MTBE approach [17] is quite non-standard and unusual. The main purpose of MTBE is automatic generation of transformation rules on a basis of an initial set of learning examples. However implementations of this approach do not guarantee that the generation of model transformation rules is correct and complete. Moreover, the generated transformation rules strongly depend on an initial set of learning examples. Current implementations of MTBE approach allow to fulfill only full equivalent mappings of attributes, disregarding the complex conversions.

In summary, it is possible to say that all considered systems have some disadvantages which restrict their applicability for transformations definitions in the MetaLanguage system. But the most appropriate and perspective, from the author's point of view, is the algebraic approach [12] with a single-pushout under condition of inclusion in it of some modifications:

- The availability of multi-level description of metamodels in the rule left- and right-hand sides.
- The description of transformation rules should be made at one level of hierarchy, and their application – on another.
- The existence of a possibility of exogenous transformations description.
- The right-hand side of production rule can contain as exposition of visual model, and some text.
- The availability of the opportunity to transform attributes of metamodel elements and constraints imposed on them.

Description of vertical transformations in MetaLanguage system has been considered explicitly in [4], therefore we will pass to reviewing of horizontal model transformations.

IV. Horizontal Model Transformations in MetaLanguage System

All horizontal transformations are described at level of metamodels that allows to specify conversions which can be applied to all models created on basis of this metamodels. For a transformation creation it is necessary to select a source and target metamodels and to define production rules that are describing conversion.

To define the rule it is necessary to select objects (entities and relations) in a source metamodel, to set constraints on pattern occurrence and to define the right-hand side of the rule. Depending on a type of transformation a right-hand side will be a text template for code generation, or a fragment of a target metamodel.

Transformation rules are applied according to their order. At first all occurrences of a first rule pattern will be found, for each of them the system will fulfill a rule right-hand side, then the system will pass to the second rule and will begin to execute it, etc.

Let's assume that the system has selected next production rule of transformation and trying to execute it. For implementation of rule application it is necessary to describe two algorithms: the algorithm of the pattern search in the source host-graph and the algorithm of execution of a right-hand side of a rule.

A. Algorithm of the Pattern Search in the Host-graph

There are various algorithms of search of subgraph isomorphic to the given pattern [18]: Ullmann algorithm, Schmidt and Druffel algorithm, Vento and Foggia algorithm, Nauty-algorithm, etc. These algorithms are the most elaborated and often used in practice.

However difference of the proposed approach from the classical task of graph matching is that in this case it is necessary to find a pattern in the metamodel graph, i.e. it is required to lead matching of graphs which belong to various hierarchy levels, thus it is necessary to consider type of nodes and arcs, as between two nodes of the metamodel graph the several arcs of various type can be led [19].

The described algorithm for finding a pattern in the graph model is a kind of backtracking algorithm that takes exponential time.

Since the amount of arcs in the model graph is less than amount of the nodes usually, each arc uniquely identifies nodes, that are incident to it, and the degree of node can be more than two, that does not allow to select the following node of the model graph, entering into a pattern, it was decided to start subgraph search in a model graph on the basis of search of particular type arcs.

At the first step of algorithm all instances of some arbitrary relation of the pattern will be found, i.e. search of an initial arc with which execution of the second step of algorithm will begin is carried out. At the second stage it is necessary to find one of possible occurrence of all relations instances of the pattern-graph G_P in the source model graph G_S. At the third step necessary nodes will be add to target graph G_T and right-hand side of the rule will be execute.

The first step is a procedure *FindPattern*:

Algorithm 1. Procedure *FindPattern*

1.1. To clear the set of the source graph nodes viewed during search – *VisitedEntities*.
1.2. To select from the pattern-graph G_P one of relations, denote it as rel. If there are not such relations, then go to the procedure *AddNodes* of adding of nodes in the graph G_T.
1.3. To find all instances of the relation rel in the source model graph G_S. The set of these instances denote as *FoundRelations*.
1.4. For each instance of the relation from the set *FoundRelations* execute procedure *FindSubGraph* to find a subgraph G_T, which corresponds to a pattern and contain the instance of relation $relI$, in the source model graph G_S.

Procedure of search of a subgraph containing the specified instance of relation *FindSubGraph* consists of following steps:

Algorithm 2. Procedure *FindSubGraph*

2.1. To add arc $relI$ to the set of arcs of the required graph G_T.
2.2. If after adding of arc it has appeared that the amount of arcs of the graph G_T equal the amount of arcs of the pattern-graph G_P then it is necessary to execute the procedure of nodes adding in the graph G_T, and then to return and remove arc $relI$ from the set of arcs of the graph G_T, since in the source graph can exist other instances of the same type relation. Otherwise, go to step 2.3.
2.3. To review the first node $entI_1$ which is incident to the arc $relI$, if it does not belong to the set *VisitedEntities*:
   a. To add the node $entI_1$ to the set *VisitedEntities*.
   b. To review all arcs of the graph G_S incoming to node $entI_1$, if the preimage some of them $fr^{-1}(rI_i^I)$ belongs to the pattern G_P and it was not considered earlier, it is necessary to search a subgraph, that contains an instance of the relation rI_i^I, starting from the second step of this algorithm.
   c. To review all arcs of the graph G_S outgoing from node $entI_1$, if the preimage some of them $fr^{-1}(rI_i^O)$ belongs to the pattern G_P and it was not considered earlier, it is necessary to search a subgraph, that contains an instance of the relation rI_i^O, starting from the second step of this algorithm.
2.4. To consider the second node $entI_2$ which is incident to the arc $relI$, if it does not belong to the set *VisitedEntities*. Reviewing is made similarly to how it has been described in step 2.3.
2.5. To execute the procedure of nodes adding to the graph G_T.

The procedure *AddNodes* of nodes adding to graph consists of three steps:

Algorithm 3. Procedure *AddNodes*

3.1. To consider all arcs of the graph G_T. If preimage of any node, that is incident to current arc, belongs to the pattern-graph G_P, it should be added to the set of nodes of the graph G_T.
3.2. To find in the graph G_S nodes, preimages of which in the graph G_P are isolated, and add them in the graph G_T.
3.3. To call the procedure of the rule right-hand side execution *ExecuteRightSide*. It is determined by a type of the transformation rule.

B. Algorithms of Rule Right-hand Side Execution

It is necessary to execute a right-hand side of production rule after the left-hand side subgraph has been found in a source graph. The algorithm of execution will depend on a type of transformation: whether transformation is a “model-model” or a “model-text”.

Transformation “model-text”. The transformation of this type allows the user to generate the source code on any target programming language on the basis of the constructed models as well as any other text representation of model, for example, its description on XML. In this case the right-hand side of production rule contains some template consisting of as static elements, which are independent of the found pattern, and dynamic parts, i.e. elements which vary depending on the found fragment of model.

For transformation fulfillment it is necessary to find all occurrences of a pattern in a source graph and to produce an insertion of an appropriate text fragment with a replacement of

a dynamic part by appropriate names of entities, relations, values of their attributes, etc.

The template is described in the target language. For selection of a dynamic part of a template the special metasymbols are used: symbol “<<” (double opening angle brackets) to indicate the beginning of a dynamic part, “>>” (double closing angle brackets) to indicate the end of a dynamic part. As entities and relations can have the same name, then for entity describing before its name the prefix “E.” is specified, and for relation describing before its name the prefix “R.” is specified.

At the transformation specifying it is possible to set constraints on pattern occurrence. These constraints allow to define the context of the rule. They contain conditions with which found fragment of model should satisfy.

Let's consider an example: define the transformation that allows on the basis of Entity-Relation Diagrams (ERD) to generate a SQL-query, building the schema of a corresponding database.

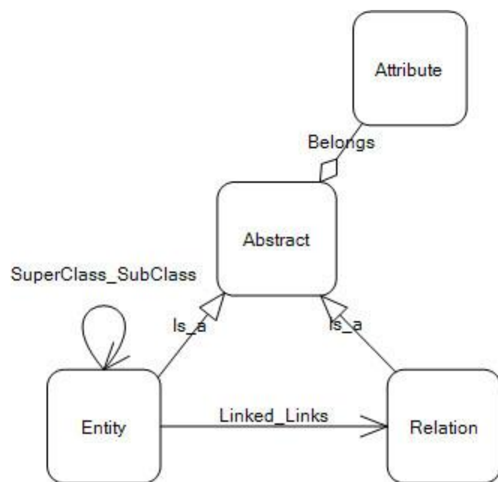


Fig. 4. Fragment of metamodel for Entity-Relation Diagrams

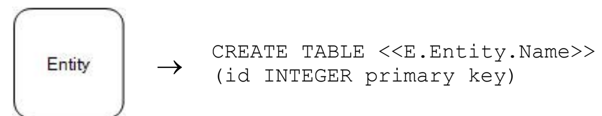
At the first step it is necessary to choose the metamodel of Entity-Relation Diagrams (see fig. 4) and to set the transformation rules. The metamodel contains the entities “Abstract”, “Attribute”, “Entity”, “Relation”. Attributes of the entity “Abstract” are “Name” that identifies an entity instance, and “Description”, containing the additional information about the entity. The entity “Abstract” is abstract, i.e. it is impossible to create instances of this entity in the model. “Abstract” acts as a parent for entities “Entity” and “Relation” (in the figure it is shown by an arrow with a triangular end). Both child entities inherit all parent attributes, relations, constraints. “Entity” does not have own attributes and constraints. “Relation” has the own attribute “Multiplicity”. The entity “Attribute” has following attributes: “Name”, “Type” and “Description”.

The bidirectional association “Linked_Links” connects entities “Relation” and “Entity”. It means that it is possible to draw equivalent relation between these entity instances in ERD-models. The second unidirectional association “SuperClass_SubClass” binds entity “Entity” with itself, it allows any instance of “Entity” to have parent (another instance of “Entity”) in ERD-models. In ERD metamodel between entities “Attribute” and “Abstract” the aggregation “Belongs” is set (in figure this relation is represented by an arc with a diamond end), therefore in ERD-models instances of entities “Relation” and “Entity” can be connected by aggregation with the instances of entity “Attribute”.

For correct transformation execution the additional attributes in the source metamodel should be added. To determine what entity is a parent, and what entity is a child it is necessary to add the mandatory attributes of a reference type “Child” and “Parent” to relation “SuperClass_SubClass”. The entity “Relation” should be transformed to the reference between relational tables, therefore we will add to “Relation” additional mandatory attributes-references of “LeftEntity” and “RightEntity” and attribute of logical type “Has_Attribute”, which will facilitate the execution of the right-hand side of production rule.

For transformation definition we will use the traditional rules of conversion of the ERD notation to a relational model, for this purpose we will define the following rules.

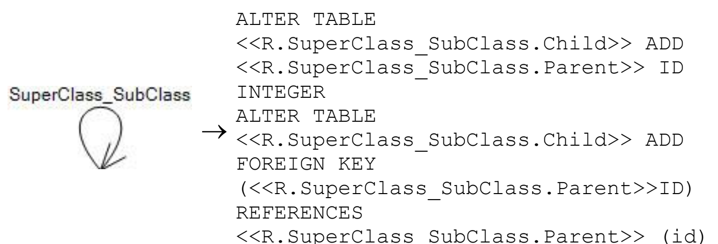
The rule “Entity” which transforms the instance of entity “Entity” to the single table looks like:




```
CREATE TABLE <<E.Entity.Name>>
(id INTEGER primary key)
```

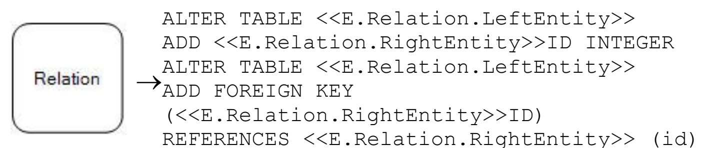
Here `<<E.Entity.Name>>` is a dynamic part of the template which allows to get a name of corresponding model entity.

As there is not inheritance relation in a relational model, it is necessary to specify the rule “Inheritance”, which for each instance of the relation “SuperClass_SubClass” in the “SubClass” table creates foreign key for connection with the “SuperClass” table. This rule looks like:




```
ALTER TABLE
<<R.SuperClass_SubClass.Child>> ADD
<<R.SuperClass_SubClass.Parent>> ID
INTEGER
ALTER TABLE
<<R.SuperClass_SubClass.Child>> ADD
FOREIGN KEY
(<<R.SuperClass_SubClass.Parent>>ID)
REFERENCES
<<R.SuperClass_SubClass.Parent>> (id)
```

The rule “Relation_1M” allows to transform instance of entity “Relation”, which does not have attributes and its multiplicity is “1:M”, to the reference between tables. The rule has the following appearance:



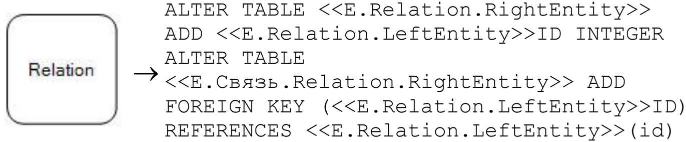

```
ALTER TABLE <<E.Relation.LeftEntity>>
ADD <<E.Relation.RightEntity>>ID INTEGER
ALTER TABLE <<E.Relation.LeftEntity>>
ADD FOREIGN KEY
(<<E.Relation.RightEntity>>ID)
REFERENCES <<E.Relation.RightEntity>> (id)
```

In this rule at first in the table corresponding to the left entity the additional column with the name `<<E.Relation.RightEntity>>ID` is added, and then the foreign key (correspondence between this additional column

and a column containing the identifiers of right table rows) is created. This rule contains the constraint on pattern occurrence:

```
E.Relation.Multiplicity = 1:M AND
E.Relation.Has_Attribute = False
```

The rule “Relation_M1” allows to transform instance of entity “Relation”, which does not have attributes and its multiplicity is “M:1”, to the reference between tables. The rule looks like:

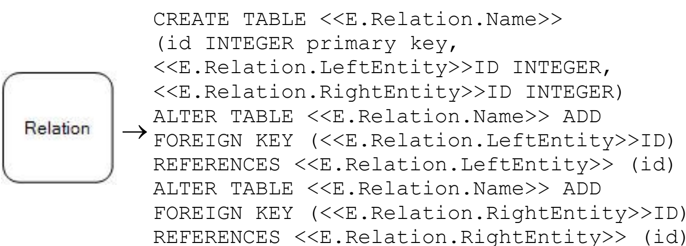



```
ALTER TABLE <<E.Relation.RightEntity>>
ADD <<E.Relation.LeftEntity>>ID INTEGER
ALTER TABLE
<<E.Связь.Relation.RightEntity>> ADD
FOREIGN KEY (<<E.Relation.LeftEntity>>ID)
REFERENCES <<E.Relation.LeftEntity>>(id)
```

The content of this rule right-hand side is similar to the content of the right-hand side of the rule “Relation_1M”. This rule contains the constraint on pattern occurrence:

```
E.Relation.Multiplicity = M:1 AND
E.Relation.Has_Attribute = False
```

For each instance of entity “Relation”, which has the attributes, or has the multiplicity “1:1” or “M:M”, it is necessary to create the single table that contains the key columns of each entity involved in relation. We call this rule “Relation_MM”, it has the following appearance:

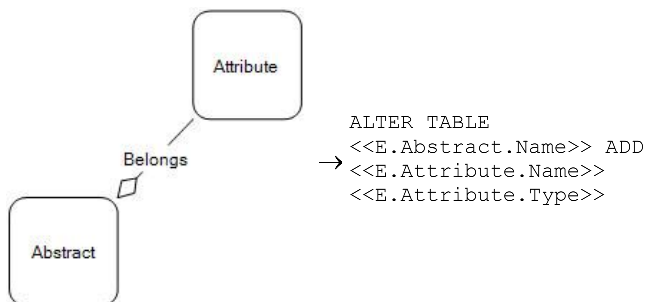



```
CREATE TABLE <<E.Relation.Name>>
(id INTEGER primary key,
<<E.Relation.LeftEntity>>ID INTEGER,
<<E.Relation.RightEntity>>ID INTEGER)
ALTER TABLE <<E.Relation.Name>> ADD
FOREIGN KEY (<<E.Relation.LeftEntity>>ID)
REFERENCES <<E.Relation.LeftEntity>> (id)
ALTER TABLE <<E.Relation.Name>> ADD
FOREIGN KEY (<<E.Relation.RightEntity>>ID)
REFERENCES <<E.Relation.RightEntity>> (id)
```

This rule contains the constraint on pattern occurrence:

```
E.Relation.Multiplicity = M:M OR
E.Relation.Multiplicity = 1:1 OR
E.Relation.Has_Attribute = True
```

The rule “Attribute” adds the columns corresponding to attributes of instances of entities and relations to the created tables:




```
ALTER TABLE
<<E.Abstract.Name>> ADD
<<E.Attribute.Name>>
<<E.Attribute.Type>>
```

Let's consider an example, apply the described transformation to the model “University” presented in fig. 5.

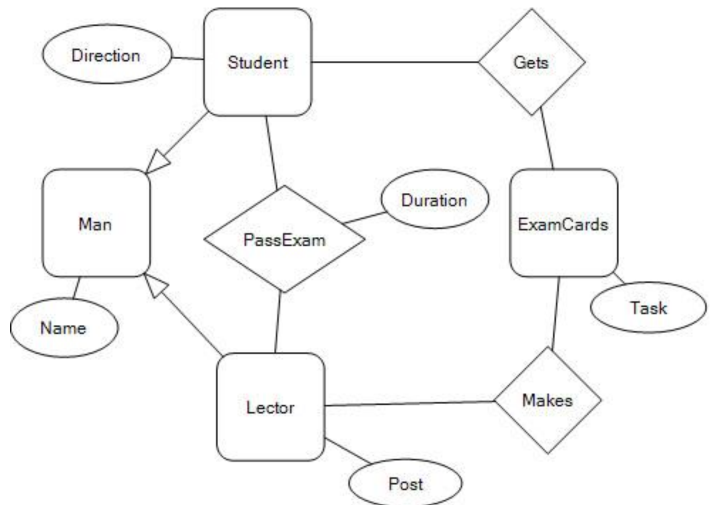


Fig. 5. Model “University” on the ERD notation

As a result the following text had been generated by the MetaLanguage system:

```
CREATE TABLE Man (id INTEGER primary key)
CREATE TABLE Student (id INTEGER primary key)
CREATE TABLE Lector (id INTEGER primary key)
CREATE TABLE ExamCards (id INTEGER primary key)
ALTER TABLE Lector ADD ExamCardsID INTEGER
ALTER TABLE Lector ADD FOREIGN KEY (ExamCardsID)
REFERENCES ExamCards (id)
ALTER TABLE ExamCards ADD StudentID INTEGER
ALTER TABLE ExamCards ADD FOREIGN KEY (StudentID)
REFERENCES Student (id)
CREATE TABLE PassExam (id INTEGER primary key,
StudentID INTEGER, LectorID INTEGER)
ALTER TABLE PassExam ADD FOREIGN KEY (StudentID)
REFERENCES Student (id)
ALTER TABLE PassExam ADD FOREIGN KEY (LectorID)
REFERENCES Lector (id)
ALTER TABLE Student ADD ManID INTEGER
ALTER TABLE Student ADD FOREIGN KEY (ManID)
REFERENCES Man (id)
ALTER TABLE Lector ADD ManID INTEGER
ALTER TABLE Lector ADD FOREIGN KEY (ManID)
REFERENCES Man (id)
ALTER TABLE Man ADD Name nvarchar(MAX)
ALTER TABLE PassExam ADD Duration nvarchar(50)
ALTER TABLE Lector ADD Post nvarchar(50)
ALTER TABLE Student ADD Direction nvarchar(MAX)
```

It should be noted that this transformation does not take into account complex conversions the ERD notation to the database schema, for example, those which would allow to create single dictionary table on the base of attribute, because it requires a special description language of templates and it is one of the areas for further research. Although such a conversion could be done by adding to the entity “Attribute” the attribute “Is_a_Dictionary” of logical type and setting the constraints on pattern occurrence.

Transformation “model-model”. Transformation of this type allows to produce conversion of model from one notation to another or to perform any operations over model (creation of new elements, reduction, etc.). Such transformation will allow to export model to external systems, and to provide the ability to convert the domain-specific language that was created by the

user in one of most common modeling language, for example, UML, ERD, IDEF0, etc.

The left-hand side of a production rule of this type transformation is a pattern, which is some fragment of the source metamodel, and the right-hand side of the rule is a some fragment of the target metamodel. At the production rule definition also it is necessary to describe the rules for converting the attributes of entities and relations. The created model should not contain dangling pointers, therefore the process of the transformation executions begins with the creation of entity instances and only then instances of relations are created. If in the process of model building the dangling pointers are still found the system will delete them.

At transformation execution it is necessary to consider the following elementary conversions:

- conversion "entity → entity";
- conversion "relation → relation";
- conversion "entity → relation";
- conversion "relation → entity".

Let's suppose that in the source model the instances of entities and/or relations of pattern are already found.

For fulfillment of the conversion $ee: Ent_L \to Ent_R$ it is necessary to create in the new model the instance $EntI_R$ of the appropriate entity of a rule right-hand side and to perform the specified transformation rules of attributes. The created instance of entity will have the same name, as the name of source entity instance.

For execution the conversion $rr: Rel_L \to Rel_R$ at first it is necessary to found in the source model the instances of entities $RelI_L.SEI$ and $RelI_L.TEI$, which are connected by the relation instance $RelI_L$, then the images of these instances $fe(RelI_L.SEI)$, $fe(RelI_L.TEI)$ should be found in the new model, and an instance of the relation from a rule right-hand side should be lead between them. After that it is necessary to fulfill transformation rules of attributes.

For fulfillment of the conversion $er: Ent_L \to Rel_R$ it is necessary to find in source model the nodes $EntI_S$, $EntI_T$ which are adjacent to entity instance $EntI_L$. Let's denote their images in the target model as *Source* and *Target*. In the target model the relation instance $RelI_R$ between nodes *Source*, *Target* should be lead. Further it is necessary to execute defined transformation rules of attributes. The algorithm of conversion $er: Ent_L \to Rel_R$ on the pseudocode can be described as follows:

Algorithm 4. Conversion "entity → relation"

$EntI_L \leftarrow$ Find_instance(Ent_L, G_S);
$EntI_S \leftarrow$ Find_adjacent_node($EntI_L$);
$EntI_T \leftarrow$ Find_adjacent_node($EntI_L$);
$Source \leftarrow$ Find_node_image($EntI_S$);
$Target \leftarrow$ Find_node_image($EntI_T$);
$RelI_R \leftarrow$ Add_new_arc($Source, Target$);
Execute_attributes_transformation($EntI_L, RelI_R$);

The complexity of the function "Find_instance" is equal to $O(N)$, where N is an amount of instances of various entities in model. The complexity of the function "Find_adjacent_node" is equal to a constant, since for its performance it is necessary to pass on the corresponding arc of the graph model. To find the image of node it is necessary to pass on arc-reference, i.e. the complexity of function "Find_node_image" is equal to a constant. The complexity of executing of function "Execute_attributes_transformation" is equal to $O(\sum_{i=1}^{k} A_i)$, where k is an amount of specified transformation rules of attributes, A_i is the complexity of the performance of i-th rule. Thus, the complexity of the presented algorithm is equal to $O(\sum_{i=1}^{k} A_i + N)$.

Conversion $re: Rel_L \to Ent_R$ transforms the instance of relation $RelI_L$ found in the source model to the entity instance $EntI_R$ of target model. For conversion execution it is necessary to create the entity instance $EntI_R$, to perform the specified transformation rules of attributes. The name of $EntI_R$ will be the same as the name of the relation instance $RelI_L$. At the next step it is necessary to find entities instances $RelI_L.SEI$, $RelI_L.TEI$, which are connected by relation instance $RelI_L$.

Further the instances of relations that connect an entity instance $EntI_R$ with nodes *Source* and *Target*, which are images of the nodes $RelI_L.SEI$ and $RelI_L.TEI$, accordingly, with keeping of orientation of relation instance.

Thus, the conversion algorithm will be following:

Algorithm 5. Conversion "relation → entity"

$RelI_L \leftarrow$ Find_instance(Rel_L, G_S);
Add_new_node($EntI_R, G_T.V$);
Execute_attributes_transformation($RelI_L, EntI_R$);
$Source \leftarrow$ Find_node_image($RelI_L.SEI$);
$Target \leftarrow$ Find_node_image($RelI_L.TEI$);
Add_new_arc($Source, EntI_R$);
Add_new_arc($EntI_R, Target$);

The complexity of algorithm of conversion "relation → entity" performance is equal to $O(\sum_{i=1}^{k} A_i + N)$.

It is possible to present the rest conversions of "model-model" type by a combination of these elementary operations.

Let's consider an example, perform the transformation of the model on Entity-Relation Diagrams notation to UML Class Diagrams.

Since the transformation is done at the metamodel level, then at the first step it is necessary to create/open source and target metamodels. The ERD metamodel was presented in the fig. 4. Metamodel of UML Class Diagrams is shown in the fig. 6. It contains the following elements: the entity "Class" and three relations "Inheritance", "Association", "Aggregation". Let's define the production rules that determine the transformation.

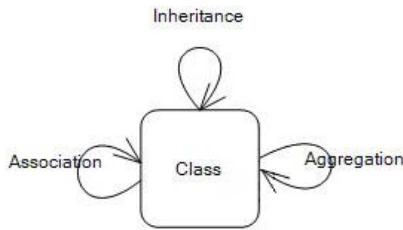
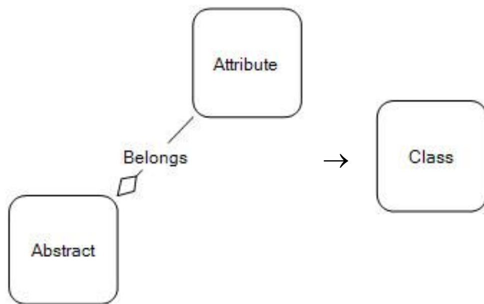
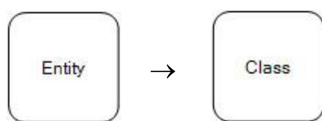


Fig. 6. Fragment of metamodel for Class Diagrams

The rule "Abstract-Class" allows to convert the instances of entities "Entity" and "Relation", which are connected at least with one instance of entity "Attribute", to the instance of entity "Class". This rule has the following appearance:

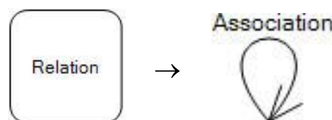


The rule "Entity-Class" allows to convert the instance of entity "Entity", which is not associated with any instance of the entity "Attribute", to the instance of an entity "Class". The rule has the following form:

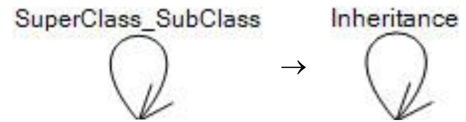


The rule "Relation-Association" converts instances of the entity "Relation" of the source model to instances of the relation "Association" of the target model.

This rule looks like:



The rule "Inheritance" puts in correspondence to each instance of the relation "SuperClass_SubClass" of source model a particular instance of the relation "Inheritance" of target model. This rule has the following form:



After definition of all rules, which are included in the transformation, it is possible to execute conversion on a specific model. Let's perform this transformation on the considered earlier model "University" (see fig. 5). The result of the transformation execution is presented in fig. 7.

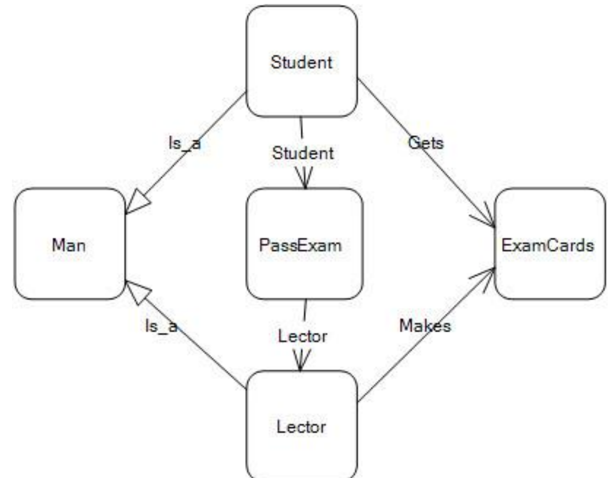


Fig. 7. Model "University" on the Class Diagrams notation, generated by MetaLanguage system

V. Conclusion

Models transformations are a central part of the model-based approach to system development, since an existence in one system of models fulfilled from the different points of view, with a different level of detail and using for the description different modeling languages, demands presence of model transformation tools both between various levels of hierarchy, and within single level: at transition from one modeling language to another.

The presented approaches have been implemented in a transformer of MetaLanguage system. This component allows to convert models, described on visual domain-specific languages, to text or other graphic models. The component has a convenient and simple user interface, therefore not only professional developers, but also domain specialists, for example, business analysts, can work with it.

References

[1] Брыксин Т.А., Литвинов Ю.В. Среда визуального программирования роботов QReal:Robots / Материалы международной конференции "Информационные технологии в образовании и науке". Самара, 2011. – С. 332-334.

[2] Демаков А. Язык описания абстрактного синтаксиса TreeDL и его использование / Препринт ИСП РАН. – 2006. – № 17. – С. 1-24.

[3] Межуев В.И. Предметно-ориентированное моделирование распределенных приложений реального времени / Системы обработки информации. – 2010. – № 5(86). – С. 98-103.

[4] Sukhov A.O., Lyadova L.N. MetaLanguage: a Tool for Creating Visual Domain-Specific Modeling Languages / Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2012). M.: Изд-во Института системного программирования РАН, 2012. – P. 42-53.

[5] Сухов А.О., Серый А.П. Использование графовых грамматик для трансформации моделей / Материалы конференции "CSEDays 2012". Екатеринбург: Изд-во Урал. ун-та, 2012. – С. 48-55.

[6] Mens T., Czarnecki K., Gorp P.V. A Taxonomy of Model Transformations / Electronic Notes in Theoretical Computer Science. Amsterdam: Elsevier Science Publishers, 2006. Vol. 152. – P. 125-142.

[7] Подкопаев А.В., Брыксин Т.А. Генерация кода на основе графической модели / Материалы межвузовского конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада "Технологии Miscrosoft в теории и практике программирования". СПб.: Изд-во СПбГПУ, 2011. – С. 112-113.

[8] Montanari U., Rossi F. Graph Rewriting, Constraint Solving and Tiles for Coordinating Distributed Systems // Applied Categorical Structures. Netherlands: Springer, 1999. – P. 333-370.

[9] Миков А.И., Борисов А.Н. Графовые грамматики в автономном мобильном компьютинге / Математика программных систем: межвуз. сб. науч. ст. / под ред. А.И. Микова, Л.Н. Лядовой. Пермь: Изд-во Перм. гос. нац. исслед. ун-та, 2012. – Вып. 9. – С. 50-59.

[10] Konig B. Analysis and Verification of Systems with Dynamically Evolving Structure / Habilitation thesis. – 238 p. [Электронный ресурс]. URL: http://jordan.inf.uni-due.de/publications/koenig/habilschrift.pdf (дата обращения: 17.02.2013).

[11] Rekers J., Schuerr A. A Graph Grammar approach to Graphical Parsing / Proceedings of the 11th IEEE International Symposium on. Washington: IEEE Computer Society, 1995. – P. 195-202.

[12] Ehrig H., Ehrig K., Prange U. et al. Fundamentals of Algebraic Graph Transformation. New York: Springer-Verlag, 2006. – 388 p.

[13] Balasubramanian D., Narayanan A., Buskirk C.P. et al. The Graph Rewriting and Transformation Language: GReAT / Electronic Communications of the EASST. – 2006. – Vol. 1. – P. 1-8.

[14] Gardner T., Griffin C., Koehler J. et al. A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard / Proccedings of the 1st International Workshop on Metamodeling for MDA. York, 2003. – P. 1-20.

[15] Csertan G., Huszerl G., Majzik I. et al. VIATRA – Visual Automated Transformations for Formal Verification and Validation of UML Models. Available at: http://static.inf.mit.bme.hu/pub/ase2002_varro.pdf (accessed 17 March 2013).

[16] ATL: Atlas Transformation Language / ATL Starter's Guide. LINA & INRIA Nantes, 2005. – 23 p.

[17] Wimmer M., Strommer M., Kargl H. et al. Towards Model Transformation Generation By-Example / Proceedings of the 40th Annual Hawaii International Conference on System Sciences. Washington: IEEE Computer Society, 2007. – P. 1-10.

[18] Серый А.П. Алгоритмы сопоставления графов для решения задач трансформации моделей на основе графовых грамматик / Математика программных систем: межвуз. сб. науч. ст. Пермь: Изд-во Перм. гос. нац. исслед. ун-та, 2012. – Вып. 9. – С. 60-73.

[19] Лядова Л.Н., Серый А.П., Сухов А.О. Подходы к описанию вертикальных и горизонтальных трансформаций метамоделей / Математика программных систем: межвуз. сб. науч. ст. Пермь: Изд-во Перм. гос. нац. исслед. ун-та, 2012. – Вып. 9. – С. 33-49.