MetaLanguage: a Tool for Creating Visual Domain-Specific Modeling Languages

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*. The technologies based on applying a metamodeling and domain-specific languages are widely used at information systems developing. There are many different tools for creating graphical domain-specific language editors with a possibility of determining user's graphical notations. However they possess disadvantages. The MetaLanguage system is designed to eliminate some of these shortcomings. MetaLanguage is a language workbench which provides creation of visual dynamic adaptable domain-specific modeling languages used in the development of information systems. In paper the approach to development of MetaLanguage DSM-platform is considered. Basic metalanguage constructions of this system are described. The mathematical multilevel domain model with usage of pseudo-metagraphs is constructed. Definitions of the graph and metagraph are given. The algorithm of vertical models transformations is described. The architecture and implementation of the development environment of MetaLanguage toolkit is presented.**

Domain-specific language; DSM-platform; MetaLanguage; metamodel; visual modeling languages; graph grammars

I. Introduction

One of the key requirements for information systems is the possibility of flexible customization to ever-changing needs of business processes and users. Domain modeling is an essential stage in the development of any information system. One of approaches for maximum adaptability – using models not only at the system development stage, but also at system functioning.

Model is an abstract description of system characteristics which are important from the viewpoint of modeling purposes. Model is described in some formal language. To each task solution can be applied a modeling language which uses concepts and relations from the information system domain. The systems life cycle is based on usage of the several models that are described from the various points of view and with different levels of abstraction. Such approach is caused by that system development process consists of several stages: analysis, design, implementation, testing. For example, at the analysis stage on the software look as on implementation of specific business functionality needed to the customer, herewith principles and details of implementation are not important.

At system creation several levels of models are created: the data that are stored in system database is a state model of the information system domain; their description, which providing a data interpretation or code generation to work with them, is a metamodel; for developing this model special formal language, which allows to work in terms of the appropriate domain, is applied – the meta-metamodel here is used.

In fact, system creation with usage of modern workbenches represents the development of domain-specific languages (DSLs) – information system meta-metamodels. DSLs are simple on applying and are easy to understand for users as they operate with domain terms. Therefore now a large number of DSLs is developed for using in different domains, for example, for business processes modeling [1] and the designing applications for mobile devices [2].

The use of DSLs and language workbenches allows to simplify process of models creation. Experts – specialists in various domains can be involved in the development. Expressiveness of languages and productivity of the systems created on their basis depends on properties of baseline models, a choice of mathematical formalism for describing language properties.

Today, there are many widespread visual DSLs, because the diagrams are more clear and understandable not only for programmers, but also for the domain experts and system users. This approach to use of visual DSLs is called domain-specific modeling (DSM). *DSM-technology* provides modeling in domain terms.

There is no unified general-purpose visual language of software development. In practice now are widely used such languages of visual modeling, as Class Diagrams and ERD – for domains modeling; IDEF, DFD, EPC, BPEL, and BPML – for business process modeling, etc.

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

Recently, UML claims to be the modeling language standard, however, this language has some significant disadvantages:

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

The language used to create other languages is called the *metalanguage*. Process of model creation can be iteratively: having created some language, we can use it as a metalanguage for designing other language which, by-turn, also can be used as a metalanguage, etc.

Despite all DSL 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. If the domain is quite simple and language is uncomplicated, the compiler will create easily. More complex domain and language will require much effort. Another shortcoming of domain-specific language is that it's necessary to create convenient graphical editors to work with it.

The *language workbench* or *DSM-platform* is the instrumental software intended to support development and maintenance of DSLs [3]. Usage at DSLs creation a language workbench considerably simplifies the process of their designing [4].

It is necessary to make following demands to tools that are using for creation of visual DSLs:

- possibility of modeling languages defining for the majority of domains, as for description of business processes, ontologies, object models, and for models of applications for mobile devices creation;
- unified representation and description of models and metamodels, i.e. for models and metamodels definition the same toolkit should be used;
- ability to dynamically change the language description without source code modification and without system restart;
- consistency of domain metamodels and models description, i.e. system should support language and models in a consistent state, and when metamodel changes system must perform all necessary modifications in corresponding models automatically;
- enabling an ability of iteration metamodels definition, i.e. describing a metamodel, the developer should be able to use it as a tool for creation other metamodels;
- possibility of models transforming from one notation to another.

II. RELATED WORKS

There are many different DSM-platforms for developing DSLs graphical editors with a possibility of determining user's notations. These tools are MetaEdit+, MS DSL Tools, Eclipse GMF, State Machine Designer, Meta Programming System, REAL-IT, UFO-toolkit, etc. A main idea of DSM-approach is to create toolkits that support optimal variants of visual modeling for specific domain. Let's consider these platforms in more detail.

UFO-toolkit [5], unlike the other systems, supports a simulation modeling of created models. This tool provides a representation of any system as a set of three-element constructions: "Unit – Function – Object" (UFO-element). The "Unit" is a point of intersection of input and output arrows. The "Function" is a transformation process of input into output. The "Object" is a substance that implements this function. The disadvantage of this system is that it does not support a possibility of models usage created in other systems since its notation does not correspond to an open standard.

Technology REAL-IT [6] is based on the use of UML. Information system development is reduced to description of the database and user interface with CASE-package REAL. On the basis of these models the application can be automatically generated. The generation possibility is provided by user interface standardization and lack of nontrivial logic of data processing. Otherwise in the generated code it is necessary to add the code written "by hand."

REAL-IT and UFO-toolkit at information systems creating allow using only the built-in modeling languages. This significantly limits the customization of these systems.

MetaEdit+ is a multiplatform environment that enables users to simultaneously work with several projects each of which can have a few models [7]. At usage this DSM-platform besides a possibility of domain-specific language creation, the developer receives the CASE tool into which this language is integrated. MetaEdit+ allows to use several DSLs at system creation.

The approach based on metamodels interpretation, instead of code generation used in MetaEdit+ allows changing the DSL definition at run-time. The system allows working with languages and metalanguages universally, using the same tools. The disadvantage of MetaEdit+ is that this DSM-platform for export of models uses an own file format (MXT) and this affects the openness of technology.

DSL Tools [8, 9] and Eclipse GMF [10, 11] technologies provide the user with advanced IDE MS Visual Studio and Eclipse, respectively. Because of this there is a possibility of code completion in high-level languages "by hand," but it can lead to inconsistency of diagrams and source code. State Machine Designer [12], in fact, is an add-on DSL Tools, eliminating some of its defect. However, the State Machine Designer allows creating a DSL only using UML Activity Diagrams that considerably limits the range of tasks.

As opposed to other DSM-platforms in the Meta Programming System [13] a method for designing textual

DSLs is supported. It's not so convenient, because the text is not sufficiently expressive.

Technology Eclipse GMF is most powerful of the above. However, its use is impeded by the lack of documentation, complexity, and frequent releases of new versions. In fact, Eclipse GMF is in a stage of intensive development.

Eclipse environment provides the user with tab GMF Dashboard which allows accelerating DSL development process by automatically generating of some language components. On GMF Dashboard tab the sequence of the operations which execution will lead to creation of a plug-in for Eclipse that allows to build diagrams in current domain is represented.

Cases when DSLs becomes part of other applications are common. For example, a specially designed language for describing business processes can be used in document circulation. Therefore one more important characteristic of the DSM-platforms is their alienability of the development environment. DSL Tools, Eclipse GMF, Meta Programming System are strongly associated with the development platforms – MS Visual Studio, Eclipse, IntelliJ-IDEA, respectively, therefore languages created by these workbenches can't be exported to external system.

All of these technologies do not provide the ability to create both visual and textual DSLs. In addition, all DSM-platforms, except for the MetaEdit+, do not allow creating the dynamic adaptability languages.

Existing problems of definition and using domain-specific visual modeling languages and DSM-platform restrictions became a reason to the MetaLanguage system creation, which would integrate the advantages of existing language workbenches and eliminate some disadvantages.

The visual metalanguage of created system should

- allow to build models that are sufficiently detailed and accurately describe the domain, so detailed and accurately how much it is necessary in each case, thus for different detail levels of description it is necessary to use the same constructions;
- have a simple constructions, allowing to work with the metalanguage not only to professional programmers but also ultimate users, such as business analysts;
- provide an opportunity to specify not only language syntax, but also its semantics.

III. Constructions of MetaLanguage System

The main shortcoming of metalanguages, which are used for DSLs designing, is their static character: the developer can't change the existing metalanguage constructions. A basis of this problem is that the metalanguage description is embedded in system source code, therefore for metalanguage modification it is necessary either to modify the source code, what to make in most cases impossible, or to offer to put up with language capabilities.

If the metalanguage description will be presented in the form of metadata, there will be possibility to change created language constructions in dynamics, i.e. without modification of system source code.

MetaLanguage system is a tool for creating visual dynamic adaptable domain-specific modeling languages used for development of information system. To describe the metamodels MetaLanguage toolkit uses metalanguage, which basic constructions are the entity, the relation, the constraint.

A. Entity

The *entity* is any construction of modeling language. Entities are characterized by

- name that uniquely identifies the entity within the metamodel;
- amount of entity instances that can be created in the model;
- set of entity attributes;
- set of entity operations;
- set of constraint imposed on the entity;
- flag of uniqueness that determines limits of entity instance name uniqueness.

The amount of entity instances defines how many instances can be created in the model. The amount of instances is set by an integer from the interval $[0, \infty)$. If value of this entity characteristic is equal to zero, then at model designing the entity of this type will not be in list of entities, proposed for creation. If the value of the characteristic is equal to infinity, it is possible to create an arbitrary number of this type entity instances.

Attribute is the named property of the entity (relation), including a description of valid values set.

The attribute has

- name that uniquely identifies it within the entity (relation);
- type that determines a set of possible values for the attribute and the operations that can be done on its values;
- default value which will be chosen as the attribute value, if the last is not specified;
- description which contains some additional information about the attribute.

Entity (relation) can have any number of attributes or not have them at all.

Operation is an abstraction of actions which can be carried out over the entity. In most cases, an applying of the operation leads to the fact that the entity changes the state.

The operation includes:

- name that uniquely identifies the operation within the entity;
- operation parameters;
- default values for parameters which in case of unavailability of basic values will be used when an operation call;
- type of returned value;
- description, containing the additional information about the operation.

Entity can have any number of operations or not have them at all.

Consider the examples of entities. Fig. 1 shows a fragment of metamodel for UML Use Case diagrams. The metamodel contains two entities “Actor” and “Use Case.”

The entity “Use Case” has following attributes: “Name,” “Description,” “Creation_Date.” The attribute “Name” has a string type and defines the Use Case name. The attribute “Description” sets the short description of the Use Case. “Creation_Date” – the attribute which contains information on when the “Use Case” has been created. Over the entity “Use Case” the following operations are admissible: “SetName(),” “SetDescription(),” “SetDate().”

An attribute of “Actor” is a string attribute “Name” which specifies the name of the actor. Permissible operation over the entity “Actor” is the “SetName()”operation.

B. Relation

Visual languages constructions in rare cases exist independently, more often they are in some way related to each other, therefore at metamodel creation importantly not only to define the basic language constructions, but also correctly specify the relations between them.

The *relation* is used for description a physical or conceptual links between entities.

Any relation is characterized by

- name that uniquely identifies the relation in this metamodel;
- type that defines the semantics of the relation;
- set of relation attributes;
- set of constraint imposed on the relation;
- multiplicity which determines how many entity instances can participate in the relation;
- flag of uniqueness that determines limits of relation instance name uniqueness.

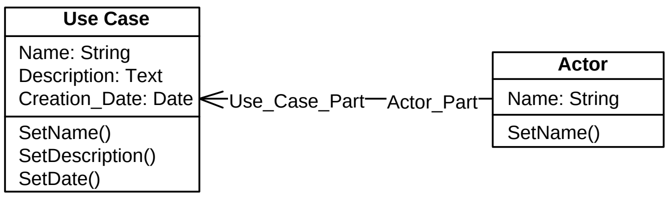


Figure 1. Fragment of metamodel for UML Use Case diagrams

The metamodel can contain the following types of relation: inheritance, association, aggregation. However in models it is possible to create only instances of the association and aggregation relations. Consider each type of relation in more detail.

Inheritance – a relation between the general entity (superclass, parent) and a specific entity (subclass, child).

The child entity inherits all parent attributes, operations and relations. In addition to the parent it can also have their own attributes, operations, relations, therefore child entity can be used everywhere where the parent entity is used, but converse is not true.

Entity can have only one parent and unlimited number of child entities, i.e. multiplicity of this type relation is 1:M.

On Fig. 2 the fragment of metamodel for Entity-Relation Diagrams is presented. The metamodel contains the entities “Abstract,” “Attribute,” “Entity,” “Relation.” In order to reduce the diagram entity operations are not represented in figure.

Attributes of the entity “Abstract” are “Name” that identifies an entity instance, and “Description,” containing the additional information about the entity.

The entity “Attribute” has following attributes: “Name,” “Type” and “Description.”

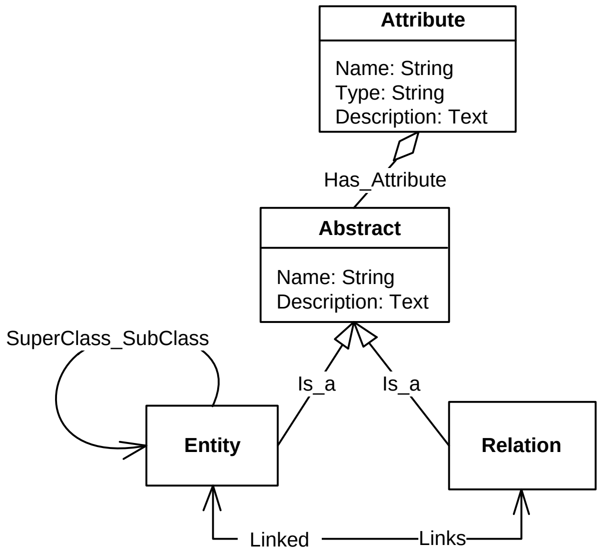


Figure 2. Fragment of metamodel for Entity-Relation Diagrams

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, operations, relations; these entities have no own attributes and operations. Entities “Relation” and “Entity” in addition to the inherited relation “Has_Attribute” have their own relation “Linked_Links.” Another association relation “SuperClass_SubClass” belongs to the entity “Entity.”

Association is a structural relationship which specifies that entities of one kind are connected to entities of another.

If two entities are connected by association, then we can navigate from one entity instances to another entity instances. The association relation can be unidirectional and bidirectional. Unidirectional association is used, when it is necessary to specify that the relation instance can be drawn only in the given direction, bidirectional association defines that the relation instance can be drawn in both directions. The case when both ends of association belong to one entity is a valid. It means that some entity instance can be associated with another instance of the same entity.

In addition to the previously described basic characteristics of the relation, there is one more which applies only to the association – a role. Entities related by association plays a role in it. The *role* is a name which uniquely identifies one of the association ends.

The arbitrary number of entity instances can participate in association as with one, and on the other hand, thus, generally a multiplicity of this relation is M:M.

On Fig. 2 two associations are presented. The bidirectional association connects entities “Relation” and “Entity” it means that in ERD-models between these entity instances it is possible to draw equivalent relation. The second unidirectional association binds entity “Entity” with itself, this allows any instance of “Entity” to have parent (another instance of “Entity”) in ERD-models.

Aggregation – a kind of association that models an unequal part-whole relation.

The main difference of aggregation from association is that the last reflects the relation between two equal entities, while in aggregation one of entities is the main and another – dependent. The distinctive features of aggregation is also the fact that this type of relation is always directed, the multiplicity of this relation is 1:M, and the aggregation ends can’t belong to one entity.

At removal of main entity instance all instances of dependent entity participating in this aggregation will be automatically deleted.

In ERD metamodel between entities “Abstract” and “Attribute” the aggregation relation 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.”

C. Constraints

In practice quite often there are cases when it is necessary to impose any constraints on entities and relations between them.

If rules of diagrams connection set syntax of visual language, constraints define its semantics. Some of constraints are set by metamodel structure, and others are described on some language. An example of the language used to describe constraints is OCL.

All constraints imposed on the metamodel can be divided into two groups: constraints imposed on the entities and constraints imposed on the relations.

Constraints imposed on the entity can be one of the following types:

- constraints imposed on the uniqueness of entity instance name;
- constraints imposed on the amount of entity instances in model;
- constraints imposed on the attribute values of entity instance.

The name of the entity instance can be unique in the metamodel, in the model or not be unique. The uniqueness in the metamodel means that in all models which are created on the basis of a current metamodel the entity instance name should be appeared only once. The constraint of such type it is necessary to set on the “Use Case” entity of metamodel for UML diagrams, if you want to specify that names of all instances of the “Use Case” entity must be unique in all models.

The uniqueness in the model means that the name of entity instance will be unique only within limits of the model of which this entity belongs. The condition of name uniqueness of the “Actor” entity in the Use Case diagram model can be an example of such constraint.

Constraint imposed on the amount of entity instances in model is set by specifying the number of instances at entity creation. So instances of abstract entities at which value of property “amount” is equal to zero, will not participate at model creation. If value of this property is equal to one, then in model it is possible to create only a single instance of this type entity. An example of this type constraint is a condition that limits an amount of created instances of the entity “Actor” by value five, it will build a clear diagram, which is not encumbered by great number of “Use Cases” and “Actors.”

In terms of defining the semantics of visual language the constraints imposed on the attribute values of entity instance are the most important. Such constraints are specified as triples:

Attribute_Name: *Sign*: *Value.*

“Value” can be a constant, attribute value of the entity instance or some function of attribute values of entity instances.

For example, in a metamodel of Use Case diagrams constraint of this type can be imposed on the attribute "Creation_Date" of the "Use Case" entity, because the date can't exceed the current time. Such constraint may look like:

$$Creation\_Date <= \text{Now}(),$$

where function Now() returns current system time.

All *constraints imposed on the relation* may be divided into following groups:

- constraints imposed on the uniqueness of relation instance name;
- constraints imposed on the types of connected entity instances;
- constraints imposed on the relations multiplicity;
- constraints imposed on the attribute values of connected entity instances.

Constraint imposed on the uniqueness of relation instance name are similar to constraint imposed on the uniqueness of entity instance name and can accept one of values: unique in the metamodel, unique in the model, non-unique.

Constraints imposed on the types of connected entity instances are defined by metamodel structure. These constraints set rules for connection of different types of entity instances. For example, the metamodel in Fig. 1 hasn't association the ends of which belong to the same entity, this means that between two instances of the "Use Case" entity or between two instances of the "Actor" entity it is impossible to create an association instance.

Constraints imposed on the relations multiplicity are set at their creation. Thus the relation of inheritance and aggregation supports only 1:M multiplicity, which can be adjusted only for dependent entity multiplicity. The association admits M:M multiplicity with the ability to refine.

If in models of Use Case diagrams it is necessary to specify that the amount of the "Actors" which involved with "Use Case" can't be more than five, then at creation of association between entities "Use Case" and "Actor" it is necessary to set the M:5 multiplicity.

The constraints imposed on the attribute values of connected entity instances carry the greatest semantic weight. Difference of these constraints from the constraints imposed on the attribute values of entity instance is that first type constraints allow setting specific entity instances on which constraints are imposed.

Constraints of this type can be set on values of attribute "Birthday" of connected entities "Person" in constructing the metamodel "Family tree," as the parent's birthday can't exceed of child's birthday.

IV. Mathematical Description of Multilevel Domain Model

Using constructions entity and relation it is possible to build any model, including an invalid in the current domain. There are various formalisms for specifying the syntax of visual languages: automatic models [14], algorithmic nets [15], graph grammars [16], et al.

Most of the existing approaches to definition visual languages syntax consider a concrete syntax, and only in rare cases – abstract syntax. The abstract syntax of visual modeling languages does not need all those details that are presented in a concrete syntax: it is possible to abstract from the choice of icons used to display the language elements, and their geometrical parameters, etc.

To define the formal rules of models creation it is proposed to use graph grammars. Graph grammar is a generalization of Chomsky grammars on graphs. To define a grammar it is required to specify the finite sets of terminal and nonterminal symbols, a finite set of production rules, and select the start symbol in nonterminal symbols set. For representation graph grammars it is necessary to choose such type of graphs which would be provided the opportunity for an iteratively metamodels definition, unified representation and description of domain models and metamodels.

Production rules in graph grammar contain the left- and the right-hand side. If to generalize the classic definition of graph grammars, then as right-hand side of the rule may be not only a labeled graph, but the code in any programming language, and also a fragment of a visual model described in other notation. That is why the graph grammar can be used for generation syntax correct models and for refactoring of existing models, code generation and model transformations from one modeling language to another [17].

As an analysis result of various representations of graph grammars it was determined that the most appropriate formalism for describing the syntax of visual modeling languages in MetaLanguage system are graph grammars, which are constructed on the pseudo-metagraphs [18]. Let's define the domain metamodel and model, applying the selected formalism, and construct the direct and reverse map of metamodel graph on model graph.

A. *Metamodel graph*

Let $Ent = \{ent_i\}, i \in \mathrm{N}, i < \infty$ (N – set of natural numbers) is a set of metamodel entities that is finite at every fixed point in time, but extends at entity creation and reduces at removing.

Let's designate each entity as a tuple

$$ent_i = \{EName_i, EICount_i, EAttr_i, EOpp_i, ERest_i, EUnique_i\},$$

where $EName_i$ is a entity name, $EICount_i$ – amount of entity instances, $EAttr_i = \{eattr_{j_i}\}, j_i \in \mathrm{N}, j_i < \infty$ – entity attributes, $EOpp_i = \{eopp_{j_i}\}, j_i \in \mathrm{N}, j_i < \infty$ – entity operations, $ERest_i = \{erest_{j_i}\}, j_i \in \mathrm{N}, j_i < \infty$ – set of constraint imposed on the entity, $EUnique_i$ – flag of uniqueness.

Sets $EAttr_i$, $EOpp_i$, $ERest_i$ are finite at every fixed point in time.

Let's divide all characteristics of i-th entity on two groups EG_i^1 and EG_i^2. The first group consists of those characteristics, which will be represented by separate nodes in graph model: sets of attributes, operations, and constraints imposed on the entity, i.e.

$$EG_i^1 = \{EAttr_i, EOpp_i, ERest_i\}.$$

Characteristics of second group $EG_i^2 = \{EName_i, EICount_i, EUnique_i\}$ (entity name, amount of entity instances, flag of uniqueness) will be attributed to node of the corresponding entity directly.

$Rel = \{rel_i\}, i \in \mathrm{N}, i < \infty$ denotes a set of metamodel relations that is finite at every fixed point in time, but extends at relation creation and reduces at removing.

Let relation is a tuple

$$rel_i = \{RName_i, RType_i, RAttr_i, RMult_i, RRest_i, RUnique_i\},$$

where $RName_i$ is a relation name, $RType_i$ – relation type, $RAttr_i = \{rattr_{j_i}\}, j_i \in \mathrm{N}, j_i < \infty$ – relation attributes, $RMult_i$ – multiplicity, $RRest_i = \{rrest_{j_i}\}, j_i \in \mathrm{N}, j_i < \infty$ – relation constraints, $RUnique_i$ – flag of uniqueness.

Sets $RAttr_i$, $RRest_i$ are finite at every fixed point in time.

Characteristics of *i*-th relation will be divided into two groups RG_i^1 and RG_i^2. The first group comprises a set of relation attributes and constraints imposed on the relation. The second group includes the following characteristics: “name,” “type,” “multiplicity,” “flag of uniqueness,” i.e.

$$RG_i^1 = \{RAttr_i, RRest_i\},$$

$$RG_i^2 = \{RName_i, RType_i, RMult_i, RUnique_i\}.$$

Consider directed pseudo-metagraph $GMM = (V, E)$. Let a set of metamodel graph nodes is a union of seven disjoint subsets:

$$V = Ent \bigcup_{i=1}^{|Ent|} EAttr_i \bigcup_{i=1}^{|Ent|} EOpp_i \bigcup_{i=1}^{|Ent|} ERest_i \bigcup$$
$$\bigcup Rel \bigcup_{i=1}^{|Rel|} RAttr_i \bigcup_{i=1}^{|Rel|} RRest_i . \quad (1)$$

The set of pseudo-metagraph arcs E divide into six disjoint subsets:

- $EEA = \{eea_i\}, i = \overline{1, |Ent|}$ – a set of arcs connecting each metamodel entity with set of attributes belonging to it;
- $EEO = \{eeo_i\}, i = \overline{1, |Ent|}$ – a set of arcs connecting each metamodel entity with set of operations over it;
- $EER = \{eer_i\}, i = \overline{1, |Ent|}$ – a set of arcs connecting each metamodel entity with set of constraints imposed on it;
- $ERA = \{era_i\}, i = \overline{1, |Rel|}$ – a set of arcs connecting each metamodel relation with set of its attributes;
- $ERR = \{err_i\}, i = \overline{1, |Rel|}$ – a set of arcs connecting each metamodel relation with set of constraints imposed on it;
- $EERR = \{eerr_i\}, i \in \mathrm{N}, i < \infty$ – a set of arcs conforming to links between entities and relations that is finite at every fixed point in time, but extends at entity (relation) creation and reduces at removing.

Thus, we see that

$$E = EEA \bigcup EEO \bigcup EER \bigcup ERA \bigcup ERR \bigcup EERR . \quad (2)$$

The *metamodel graph* is a directed pseudo-metagraph $GMM = (V, E)$, for which (1) and (2), where V is a nonempty set of graph nodes, E is a set of graph arcs.

Let's consider an example. We will construct a metamodel graph for the entity “Use Case” of UML Use Case diagrams. Metamodel of this diagram type is shown in Fig 1. Attributes of the entity “Use Case” are “Name,” “Description,” “Creation_Date.” Operations that can be performed on entity – “SetName(),” “SetDescription(),” “SetDate(),” i.e. for given entity

$$EAttr_i = \{\text{“}Name\text{,” “}Description\text{,” “}Creation\_Date\text{”}\},$$

$$EOpp_i = \{\text{“}SetName()\text{,” “}SetDescription()\text{,” “}SetDate()\text{”}\},$$

$$ERest_i = \varnothing .$$

The metamodel graph corresponding to a fragment of the “Use Case” entity shown in Fig. 3.

As can be seen from figure

$$EEA = \{eea_{i_1}\},\ EEO = \{eeo_{i_1}\},\ EER = \varnothing,\ EERR = \varnothing .$$

B. *Model Graph*

The *model* is actually an “instance” of metamodel in which:

- the attributes of entity – a concrete values;
- there are no operations over entity instances and constraints imposed on the entity and relation instances;
- inheritance relation instances can’t be created.

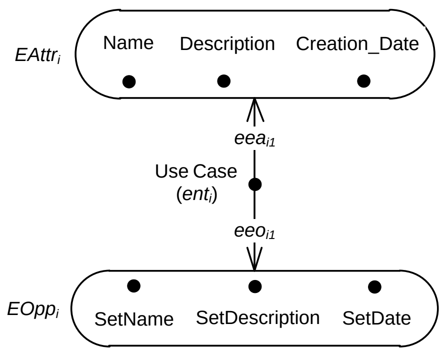


Figure 3. Fragment of metamodel graph for “Use Case” entity

Let's designate a set of all models which have been created based on the current metamodel through $M = \{m_k\}, k \in \mathrm{N}, k < \infty$ that is finite at every fixed point in time, but extends at model creation and reduces at removing.

Let's introduce following notation:

- $EntI_i$ – set of instances of *i*-th entity;
- $EAttrI_{j_i}$ – set of attribute values for *j*-th instance of *i*-th entity;
- $RelI_k$ – set of instances of *k*-th relation;
- $RAttrI_{k_l}$ – set of attribute values for *k*-th instance of *l*-th relation.

Sets $EntI_i$, $EAttrI_{j_i}$, $RelI_k$, $RAttrI_{k_l}$ are finite at every fixed point in time, but extend at entity (relation) instance creation and reduce at removing.

Examine the directed pseudo-metagraph $GM = (VI, EI)$. Let a set of model graph nodes is a union

$$VI = \bigcup_{i=1}^{|Ent|}\left(EntI_i \bigcup_{j=1}^{|EAttr_i|} EAttrI_{j_i} \right) \bigcup_{k=1}^{|Rel|}\left(RelI_k \bigcup_{l=1}^{|RAttr_k|} RAttrI_{l_k} \right). \quad (3)$$

Consider the following example. Let’s create a model graph for instance of “Use Case” entity (Fig. 4).

From a figure it is apparently that $EAttrI_i$ = {“*Pass_exam*,” “*Use Case describes passing an exam process*,” “21/06/09”}.

The set EI divides into three disjoint subsets:

- $EEAI = \{eeaI_i\}, i = \overline{1, |EntI|}$ – a set of arcs connecting each entity instance with set of attributes belonging to it;
- $ERAI = \{eraI_i\}, i = \overline{1, |RelI|}$ – a set of arcs connecting each relation instance with set of attributes belonging to it;
- $EERRI = \{eerrI_i\}, i \in \mathrm{N}, i < \infty$ – a set of arcs corresponding to the links between entity instances and relation instances that is finite at every fixed point in time, but extends at entity (relation) instance creation and reduces at removing.

Thus, we see that

$$EI = EEAI \bigcup ERAI \bigcup EERRI. \quad (4)$$

You can see from the Fig. 4 that for represented “Use Case” entity instance $EEAI = \{eeaI_{i_1}\}$, $EERRI = \varnothing$.

The *model graph* is a directed pseudo-metagraph $GM = (VI, EI)$, for which (3) and (4), where VI is a nonempty set of graph nodes, EI – set of graph arcs.

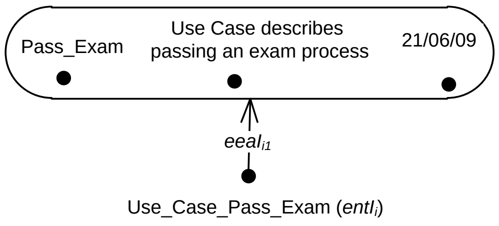

Pass_Exam
Use Case describes passing an exam process
21/06/09
eeaI_i1
Use_Case_Pass_Exam (entI_i)

Figure 4. Model graph corresponding to “Use Case” entity instance

C. Operation of Model Graph Creation

Let's construct map of the metamodel graph on the model graph, it corresponds to an operation of a model graph creation. Such map allow to support models in an actual state, as metamodel modification leads to a change of all models created based on it.

Let's introduce following notation:

- $EntI = \bigcup_{i=1}^{|Ent|} EntI_i$ – a set of model graph nodes conforming to all entity instances;
- $RelI = \bigcup_{i=1}^{|Rel|} RelI_i$ – a set of model graph nodes corresponding to all relation instances;
- $EAttrI = \bigcup_{i=1}^{|Ent|} \bigcup_{j_i=1}^{|EntI|} EAttrI_{j_i}$ – a set of model graph nodes conforming to attribute values of all entity instances;
- $RAttrI = \bigcup_{k=1}^{|Rel|} \bigcup_{l_k=1}^{|RelI|} RAttrI_{l_k}$ – a set of model graph nodes corresponding to attribute values of all relation instances.

Sets $EntI$, $RelI$, $EAttrI$, $RAttrI$ are finite at every fixed point in time, but extend at entity (relation) instance creation and reduce at removing.

Let’s construct a map that for each metamodel graph entity-node defines a set of model graph nodes conforming to instances of this entity, i.e.

- $(\exists\, ent_i \in Ent)(\exists\, entI_{j_i} \in EntI) : fe(ent_i) = entI_{j_i}$, if entity is not abstract and has instances;
- $(\exists\, ent_i \in Ent) : fe(ent_i) = \varnothing$, if entity is abstract and does not have instances.

Map fe defines creation operation of node corresponding to entity instance.

Let's define map of metamodel graph nodes $EAttr$ conforming to a set of entity attributes on a set of model graph nodes $EAttrI$:

$$fea : EAttr \to EAttrI.$$

And besides

$$(\forall eattr_{j_i} \in EAttr)(\exists\, eattrI_{k_{j_i}} \in EAttrI):\ fea(eattr_{j_i}) = eattrI_{k_{j_i}},$$
$$i = \overline{1,|Ent|},\ j_i = \overline{1,|EntI_i|},\ k_{j_i} = \overline{1,|EAttrI_{j_i}|}.$$

Map fea corresponds to the operation of assignment a value to entity instance attribute.

Let's examine a set of metamodel graph nodes which correspond to relations. With each node we associate a set of graph model nodes that appropriate to particular relation instances, as a result we obtain a map $fr : Rel \to RelI$, such that the following

- $(\exists\, rel_i \in Rel)(\exists\, reli_{j_i} \in RelI):\ fr(rel_i) = reli_{j_i}$, if relation has instances;
- $(\exists\, rel_i \in Rel):\ fr(rel_i) = \varnothing$, if relation does not have instances.

This map defines creation operation of node corresponding to relation instance.

Let's define operation of assignment a value to relation instance attribute. To do this, we will construct a map of metamodel graph nodes $RAttr$ conforming to a set of relation attributes on set of model graph nodes corresponding to attribute values $RAttrI$: $fra : RAttr \to RAttrI$.

And besides

$$(\forall rattr_{j_i} \in RAttr)(\exists\, rattrI_{k_{j_i}} \in RAttrI):\ fra(rattr_{j_i}) = rattrI_{k_{j_i}},$$
$$i = \overline{1,|Rel|},\ j_i = \overline{1,|RelI_i|},\ k_{j_i} = \overline{1,|RAttrI_{j_i}|}.$$

Thus, maps $fe,\ fea,\ fr,\ fra$ define matching between set of metamodel graph nodes and set of model graph nodes (Fig. 5).

Now we will define the rules under which the arcs of graph GMM are mapped to the arcs of graph GM.

Let's construct the map $gea : EEA \to EEAI$, according to which each arc of the set EEA is put in correspondence with specified arcs of the set $EEAI$, i.e.

$$(\forall eea_{j_i} \in EEA)(\exists\, eeaI_{k_{j_i}} \in EEAI) : gea(eea_{j_i}) = eeaI_{k_{j_i}},$$
$$i = \overline{1,|Ent|},\ j_i = \overline{1,|EntI_i|},\ k_{j_i} = \overline{1,|EAttrI_{j_i}|}.$$

Similarly, we can define a map $gra : ERA \to ERAI$ for which

$$(\forall era_{j_i} \in ERA)(\exists\, eraI_{k_{j_i}} \in ERAI) : gra(era_{j_i}) = eraI_{k_{j_i}},$$
$$i = \overline{1,|Rel|},\ j_i = \overline{1,|RelI_i|},\ k_{j_i} = \overline{1,|RAttrI_{j_i}|}.$$

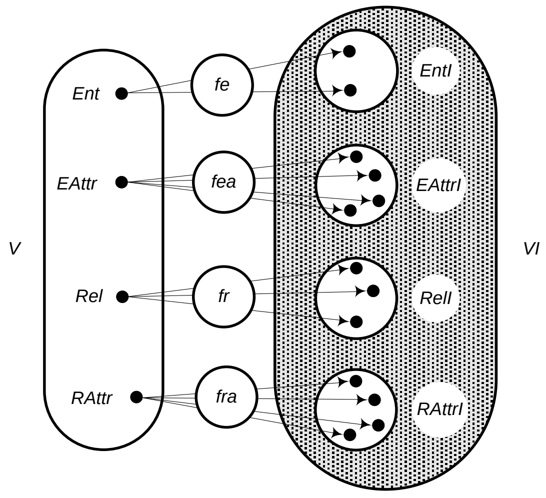


Figure 5. The map of metamodel graph nodes on model graph nodes

Let's construct the map $ger : EERR \to EERRI$, according to which each arc of the set $EERR$ is put in correspondence with specified arcs of the set $EERRI$, i.e.

$$(\forall eerr_i \in EERR)(\exists\, eerrI_{k_{j_i}} \in EERRI) : ger(eerrI_{k_{j_i}}) = eerr_i,$$
$$i = \overline{1,|Ent|},\ j_i = \overline{1,|EntI_i|},\ k_{j_i} = \overline{1,|EAttrI_{j_i}|}.$$

Thus, maps gea, gra, ger define matching between the set of metamodel graph arcs and the set of model graph arcs.

Model graph creation is a map of metamodel graph on model graph at which conversions are performed $fe,\ fea,\ fr,\ fra, gea, gra, ger$.

D. *Operation of Model Interpretation*

Let's construct map of model graph on metamodel graph. It defines operation of model interpretation which allows to execute operations over entity instances and to check constraints imposed on the entities and relations.

As model graph nodes are instances of metamodel graph nodes, it is possible to define the map of the model graph nodes on the metamodel graph nodes.

Let's construct a surjection $fe^{-1} : EntI \to Ent$ which to each model entity instance puts in correspondence metamodel entity

$$(\forall entI_{j_i} \in EntI)(\exists !ent_i \in Ent):\ fe^{-1}(entI_{j_i}) = ent_i,$$
$$i = \overline{1,|Ent|},\ j_i = \overline{1,|EntI_i|},$$

and besides several elements of the set $EntI$ may correspond to one entity, i.e. is performed

$$(\forall ent_i \in Ent)(\exists\, entI_{j_i}, entI_{k_i} \in EntI, entI_{j_i} \neq entI_{k_i}):$$
$$fe^{-1}(entI_{j_i}) = fe^{-1}(entI_{k_i}) = ent_i.$$

Let's define map which is an inverse of map fea :

$$fea^{-1} : EAttrI \to EAttr .$$

This surjection to each element of set $EAttrI$ puts in correspondence a unique element of set $EAttr$, i.e.

$$(\forall eattrI_{k_{j_i}} \in EAttrI)(\exists ! eattr_{j_i} \in EAttr) : fea^{-1}(eattrI_{k_{j_i}}) =$$
$$= eattr_{j_i}, i = \overline{1, |Ent|},\ j_i = \overline{1, |EntI_i|},\ k_{j_i} = \overline{1, |EAttrI_{j_i}|} ,$$

and besides several elements of the set $EAttrI$ may correspond to one element of the set $EAttr$, i.e. is performed

$$(\forall ea_{j_i} \in EAttr)(\exists\, eaI_{k_{j_i}}, eaI_{l_{j_i}} \in EAttrI, eaI_{k_{j_i}} \ne eaI_{l_{j_i}}) :$$
$$fea^{-1}(eaI_{k_{j_i}}) = fea^{-1}(eaI_{l_{j_i}}) = ea_{j_i} .$$

Let's consider a set of model graph nodes that correspond to relation instances. Each such node we associate with a unique metamodel graph node, which corresponds to a current relation, as a result we obtain a surjective map $fr^{-1} : RelI \to Rel$ for which

$$(\forall relI_{j_i} \in RelI)(\exists ! rel_i \in Rel) : fr^{-1}(relI_{j_i}) = rel_i ,$$
$$i = \overline{1, |Rel|},\ j_i = \overline{1, |RelI_i|} ,$$

and multiple relation instances may be created on the basis of one relation, i.e. is performed

$$(\forall rel_i \in Rel)(\exists\, relI_{j_i}, relI_{k_i} \in RelI, relI_{j_i} \ne relI_{k_i}) :$$
$$fr^{-1}(relI_{j_i}) = fr^{-1}(relI_{k_i}) = rel_i .$$

Surjective map $fra^{-1} : RAttrI \to RAttr$ which is an inverse of map fra , each model node conforming to relation attribute value associates with a unique metamodel node from set $RAttr$:

$$(\forall rattrI_{k_{j_i}} \in RAttrI)(\exists ! rattr_{j_i} \in RAttr) : fra^{-1}(rattrI_{k_{j_i}}) =$$
$$= rattr_{j_i}, i = \overline{1, |Rel|},\ j_i = \overline{1, |RelI_i|},\ k_{j_i} = \overline{1, |RAttrI_{j_i}|} ,$$

and multiple elements of the set $RAttrI$ may correspond to one element of set $RAttr$, i.e. is performed

$$(\forall ra_{j_i} \in RAttr)(\exists\, raI_{k_{j_i}}, raI_{l_{j_i}} \in RAttrI, raI_{k_{j_i}} \ne raI_{l_{j_i}}) :$$
$$fra^{-1}(raI_{k_{j_i}}) = fra^{-1}(raI_{l_{j_i}}) = ra_{j_i} .$$

Thus, four maps $fe^{-1}, fea^{-1}, fr^{-1}, fra^{-1}$ define matching between the set of model graph nodes and the set of metamodel graph nodes (see Fig. 6).

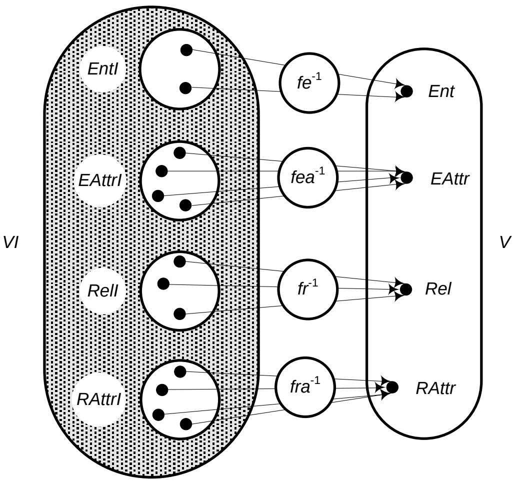


Figure 6. The map of model graph nodes on metamodel graph nodes

Since operations over entity and relation instances are not defined, then for navigation between the entities, relations and their instances let's extend set of model graph arcs with the arc-references connecting entity and relation instances with those metamodel entities and relations on which basis they are created. Let's denote the set of such arcs through

$$T = \bigcup_{i=1}^{|Ent|+|Rel|} T_i ,\ T_i = \{t_{j_i}\},\ j = \overline{1, |EntI_i| + |RelI_i|} .$$

Now we will define the rules under which the arcs of model graph GM are associated with the arcs of metamodel graph GMM .

Let's construct the map $gea^{-1} : EEAI \to EEA$ which to each arc of the set $EEAI$ puts in correspondence unique arc of the set EEA, i.e.

$$(\forall eeaI_{k_{j_i}} \in EEAI)(\exists ! eea_{j_i} \in EEA) : gea^{-1}(eeaI_{k_{j_i}}) = eea_{j_i} ,$$
$$i = \overline{1, |Ent|},\ j_i = \overline{1, |EntI_i|},\ k_{j_i} = \overline{1, |AttrI_{j_i}|} .$$

Similarly, we can define a map $gra^{-1} : ERAI \to ERA$ for which

$$(\forall eraI_{k_{j_i}} \in ERAI)(\exists ! era_{j_i} \in ERA) : gra^{-1}(eraI_{k_{j_i}}) = era_{j_i} ,$$
$$i = \overline{1, |Rel|},\ j_i = \overline{1, |RelI_i|},\ k_{j_i} = \overline{1, |AttrI_{j_i}|} .$$

As can be seen from definition the maps gea^{-1} and gra^{-1} are surjective.

Let's construct the surjective map $ger^{-1} : EERRI \to EERR$ which to each arc of the set $EERRI$ puts in correspondence unique arc of the set $EERR$, i.e.

$$(\forall eerrI_{j_i} \in EERRI)(\exists ! eerr_i \in EERR) : ger^{-1}(eerrI_{j_i}) = eerr_i ,$$
$$i = \overline{1, |EERR|},\ j_i = \overline{1, |EERRI_i|} .$$

Thus, maps $gea^{-1}, gra^{-1}, ger^{-1}$ define single-valued transformation between set of model graph arcs and set of metamodel graph arcs.

Model interpretation is a map of model graph on metamodel graph at which conversions are performed $fe^{-1}, fea^{-1}, fr^{-1}, fra^{-1}, gea^{-1}, gra^{-1}, ger^{-1}$.

V. Development Environment of Metalanguage System

To work with metalanguage objects the development environment that includes the following components: graphical editor, object browser, repository, validator, generator is designed.

The *development environment* includes implementation of the general service functions of created system. It integrates all components into a single unit.

Graphic Editor – a work area for drawing diagrams. Assignment of the Editor is a creation, modification, removal of models, and also establishment of links between different models. Each model entity is represented by some graphic symbol, and relations between entities are represented by different types of lines.

The Graphical Editor allows to allocate on a worksheet various shapes (instances of entities and relations), to apply to these shapes different actions, to set various graphical properties for them.

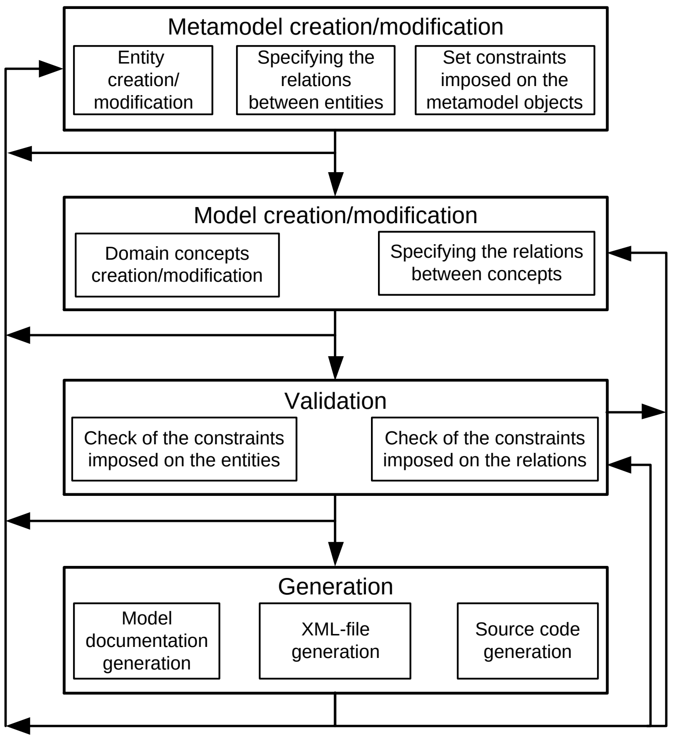
Object Browser – a tool designed for viewing and editing information stored in the repository. The browser provides the ability to export/import models to/from external systems. A format for models import/export is the XML which contains besides the data also metadata that describe structure of the stored information.

Uniform storage of all information about the system is the *repository*. It contains the information about metamodels, models, entities, relations, attributes, constraints, icons used to image entities and relations. Repository stores the information about models as well as metamodels uniformly it allows to process them with a single tool. Physically, the repository is a relational database.

The *Validator* checks correspondence of model to the constraints specified by the user. At check each constraint will be applied to each instance of entities and relations. If constraint is not performed, the error message will be shown.

The *Generator* allows generating XML-file, model documentation or source code on the basis of existing models. XML-file will contain information about the model: model properties, entities, relations, their attributes, constraints imposed on the model. Model documentation includes: model name, information about developers who took part in its creation, graphical representation of model with links to description of its individual parts.

Having described the basic components of a MetaLanguage system, let consider how visual domain-specific modeling languages are designed (Fig. 7).



Metamodel creation/modification
Entity creation/ modification
Specifying the relations between entities
Set constraints imposed on the metamodel objects
Model creation/modification
Domain concepts creation/modification
Specifying the relations between concepts
Validation
Check of the constraints imposed on the entities
Check of the constraints imposed on the relations
Generation
Model documentation generation
XML-file generation
Source code generation

Figure 7. Process of creation/modification the by means of MetaLanguage system

Process of DSL definition begins with metamodel creation. For this purpose it is necessary to specify the main constructions of created language, to define relations between them, to set constraints imposed on the metamodel entities and relations. After building of metamodel the developer gets a customizable extensible visual modeling language.

Using created DSL, the user can design models containing objects that describe specific domain concepts and links between them.

The Validator should check up whether model satisfies to constraints which were imposed on it after model constructing.

Using the Generator, the developer can save the constructed metamodels and models in the form of XML-files or generate system documentation or source code based on them.

Note that at metamodel modification the system automatically will make all necessary changes in the models which are created on the basis of this metamodel.

VI. Conclusion and Future Works

The article describes the language workbench MetaLanguage which can be used at all stages of information system creation from domain-specific modeling languages development to creating of models that used in a particular system implementation or for source code generation.

The analysis of existing analogues has shown that there are unresolved problems: impossibility of export of DSLs and models to external systems, impossibility of models transformations from one notation to another, impossibility of dynamic adaptability of languages. It was decided to eliminate these DSM-platforms restrictions at MetaLanguage system engineering.

The development environment is simple to use, therefore not only professional programmers, but also domain experts, for example, business analysts, can work with this toolkits. Thus the developer gets powerful workbench for creation of visual dynamic adaptable domain-specific modeling languages.

To work with models and metamodels uniformly, it is used the same tools, therefore process of model creation can be iterative.

Metamodels modification can be made at any stage of DSL creation. Thus after metamodel modification the system automatically will make all necessary changes in models which are created on basis of this metamodel.

For unified models creation the mathematical model – graph grammars based on pseudo-metagraphs – was constructed. This formalism has allowed to describe basic elements and algorithms which MetaLanguage uses in its work: algorithms for creation/modification of domain metamodels and models, algorithms for vertical models transformation, algorithms for constraint checking.

The paper also presents the approaches to implementation of metalanguage and development environment to work with it. This environment allows to create modeling languages that

- can be flexibly configured not only to ever-changing needs of business processes and users, but also to other domains;
- provide an opportunity to work in domain terms;
- have a high degree of consistency with the metalanguage;
- can be reused in similar projects.

The research prototype of MetaLanguage system that implements the functionality described above was created in the present time. In the future it is planned to continue working in this direction:

- to design the DSLs for various purposes, for example, for description of ontologies, document templates, business processes with created DSM-platform;
- to describe algorithms for the horizontal transformation of graph representation which will allow to make transformation of domain models from one notation to another;
- to integrate language workbench MetaLanguage with some CASE tool that allows to develop information systems, for example, with METAS CASE system [19].

References

[1] А.О. Сухов, Л.Н. Лядова "Использование визуальных предметно-ориентированных языков для описания бизнес-процессов", Материалы межвуз. конкурса-конференции "Технологии Microsoft в теории и практике программирования", СПб, 2009. С. 117.

[2] А.О. Сухов "Использование предметно-ориентированных языков при создании приложений для мобильных устройств", Материалы всероссийской научно-практической конференции студентов "Студент и наука", т. 3, 2010. С. 75-76.

[3] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?" Available at: http://martinfowler.com/articles/languageWorkbench.html (accessed 10 April 2012).

[4] Л.Н. Лядова, А.О. Сухов "Визуальные языки и языковые инструментарии: методы и средства реализации", Труды международных научно-технических конференций "Интеллектуальные системы" (AIS'10) и "Интеллектуальные САПР" (CAD-2010), т. 1, 2010. С. 374-382.

[5] В.С. Маторин "CASE-инструментарий UFO-toolkit. Автоматизация построения УФО-моделей", Проблемы программирования, №2, 2004. – С. 144-149.

[6] А.Н. Иванов "Технологическое решение REAL-IT: создание информационных систем на основе визуального моделирования", Сб. "Системное программирование" под ред. проф. А.Н.Терехова и Д.Ю.Булычева, 2004. – С.89-100.

[7] J.-P. Tolvanen, M. Rossi, "MetaEdit+: defining and using domain-specific modeling languages and code generators." Available at: http://portal.acm.org/citation.cfm?id=949365 (accessed 10 April 2012).

[8] S. Cook, G. Jones, S. Kent, A.C. Wills, "Domain-Specific Development with Visual Studio DSL Tools," Reading: Addison-Wesley, 2007.

[9] "Creating Domain-Specific Languages." Available at: http://msdn.microsoft.com/en-us/library/bb126259(v=vs.80).aspx (accessed 10 April 2012).

[10] R.C. Gronback "Eclipse Modeling Project: A Domain-Specific Language Toolkit," Reading: Addison-Wesley, 2009.

[11] T. Ozgur, "Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling In the context of the Model-Driven Development." Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.6383&rep=rep1&type=pdf (accessed 10 April 2012).

[12] А.В. Ларионов "Разработка визуального языка автоматного программирования". Available at: http://is.ifmo.ru/papers/StateMachineDesigner.pdf (accessed 10 April 2012).

[13] S. Dmitriev, "Language Oriented Programming: The Next Programming Paradigm." Available at: http://www.onboard.jetbrains.com/is1/articles/04/10/lop/index.html (accessed 10 April 2012).

[14] А.П. Стасенко "Автоматная модель визуального описания синтаксического разбора", Вычислительные технологии, вып. 5, т. 13, 2008. – С. 70-87.

[15] О.Ф. Королев "Алгоритмические сети как визуальный язык программирования", Труды СПИИРАН, вып. 2, 2005. – С. 130-137.

[16] J. Rekers, A. Schuerr, "A Graph Grammar approach to Graphical Parsing," Visual Languages Proceedings, 11th IEEE International Symposium, Darmstadt, pp. 195-202, 1995.

[17] B. Courcelle, "Graph Rewriting: An Algebraic and Logic Approach," Handbook of Theoretical Computer Science, vol. B, 1990, pp. 193-242.

[18] А.О. Сухов "Анализ формализмов описания визуальных языков моделирования", Современные проблемы науки и образования, №2, 2012. Available at: http://www.science-education.ru/102-5655 (accessed 10 April 2012).

[19] Л.Н. Лядова, С.А. Рыжков "CASE-технология METAS", Математика программных систем, Межвуз. сб. науч. статей, Пермь, 2003. С. 4-18.