Complex Model Transformation within MDA Approach

Evgeny Cherkashin[1,2], Alexey Shigarov[1], Viacheslav Paramonov[1], and Ljubica Kazi[3]

[1] Matrosov Institute for System Dynamics and Control Theory of SB RAS,
134 Lermontov Street, Irkutsk, 664033, Russia
[2] National Research Irkutsk State Technical University,
83 Lermontov Street, Irkutsk, 664074, Russia
University of Novi Sad, Technical faculty ”Mihajlo Pupin”,
Đure Đakovića bb, Zrenjanin, 23000, Serbia
[3] `{eugeneai,shig,slv}@icc.ru, ljubica.kazi@gmail.com`

Abstract. The problem of software modeling and its transformation using complex approach, having various models as sources, is considered. The software system (most frequently, information system) is synthesized as a result of a logical inference of a set of subgoals (a scenario) represented in the LogTalk programming language within Model Driven Architecture (MDA) paradigm. The source model is represented as a set of BPMN2.0, SysML, CMMN and UML diagrams. The models are converted in a A-box and stored on an ontology server. The models are transformed into Platform Specific Models. Additional data for the transformation are provided from Linked Open Data compliant sources. Usage of such kind of MDA allows us to develop information systems on the abstract model level, refining the source model with modeling instruments specially intended for the corresponding modeling aspect.

Keywords: model driven architecture, logical inference, complex model of software, linked open data

1 Introduction

The simplest approach to a system improvement consists of the repetitive execution of the following steps over a level of organization under reingeneering:

1. imaging the target state of the system (future),
2. comparison with the present state,
3. assessment of the deviations from the target state,
4. planning set of actions for reaching the target state,
5. performing the planned set of actions,
6. control of the achieving the target state by, e.g., checking the set of target criteria.

Business process (BP) reingeneering usually starts from a technical audit (TA), the process of a system analysis of an organization structure, its problems and resources [1]. The result of the TA is set of text documents describing the present state of the organization and its BPs on the various levels (social, administrative, technical, and physical). The set comprises specifications of organization structure, BP models (AS-IS, TO-BE), target state requirements and restrictions, etc.

[[[Point of views: people resources, marketing, technology, IT, finance]]]

There are three general creative roles of stakeholders involved in the improvement and revealed during TA: *problem owner*, *knowledge owner*, and *problem solver*. Problem owner is the agent (person), who recognizes the problems in the current state. This role describes future target state as a ranged list of *requirements* and *restrictions*. Knowledge owner describes the states in technical terms of the problem domain. The description contains structure of BPs, their sequences, parameters, causal relationship, control processes, etc. Problem solver searches and devices a problem solution, relying on a problem solving methodology, respecting interests of the problem owner, and information acquired from knowledge owner.

[[[How we can represent specifications as IT SUBSYSTEM of an existing system? dring the process of improvement]]]

[[[Application of the nowadays modeling notations in specifications for the low levels]]] .....MOD to be improved with more expressive but complex notions of SysML (System Markup Language), BPMN–2.0 (Business Process Modeling Notation) and CMMN (Case Management Modeling Notation). SysML used to describe organizational and administrative structures in higher degree of abstraction with respect to UML2, which is used at administrative and technical levels, and in the same time to be more detailed. The usage of these modeling notation will allow to describe the present organizational structure more formally and precisely as a Computational Independent Model (CIM) and Platform Independent Model (PSM) of MDA paradigm, resulting the bridge between specifications and IT resource under development.

Heaving described BPs structure and flows of an organization in the formal detailed multilevel form in SysML, we could construct an environment of a permanent IS research and development. At the first stage, a most simple case is being realized: the description could be converted into check lists controlling conditions of business process starting and ending points. The second stage of implementation is the automation of planning after reaching a checked condition. The third stage is to organize data processing according to the synthesized plan. In the long run a refined specification description is being integrated with already used software. All the stages are accompanied with a lot of useful information to be accumulated, forming the quality measurement, knowledge and analysis basis [**?**].

The *object* of the research is the development of a generative technology [] of expression of process structure of an organization in abovementioned notations (SysML, BPMN-2.0, CMMN, UML) and conversions of the description into subsystems of a complex information system supporting the BP reingeneering processes. The *subject* we consider in this paper is to describe tools and techniques of complex model definition and its interpretation as a MDA transformations.

2 Architecture of MDA tools

The architecture of the software development tools is based on the provision of uniformity of the transformation rules language and predicate input data representation. That's why we require either to store all the input model data in Semantic Web graphs of ontologies, or provide an interface representing the source model as the ontology graph. The ontologies could be served as RDF–files of triples or with servers having SPARQL endpoints. A general architecture of the development tools is presented in Fig. 1.

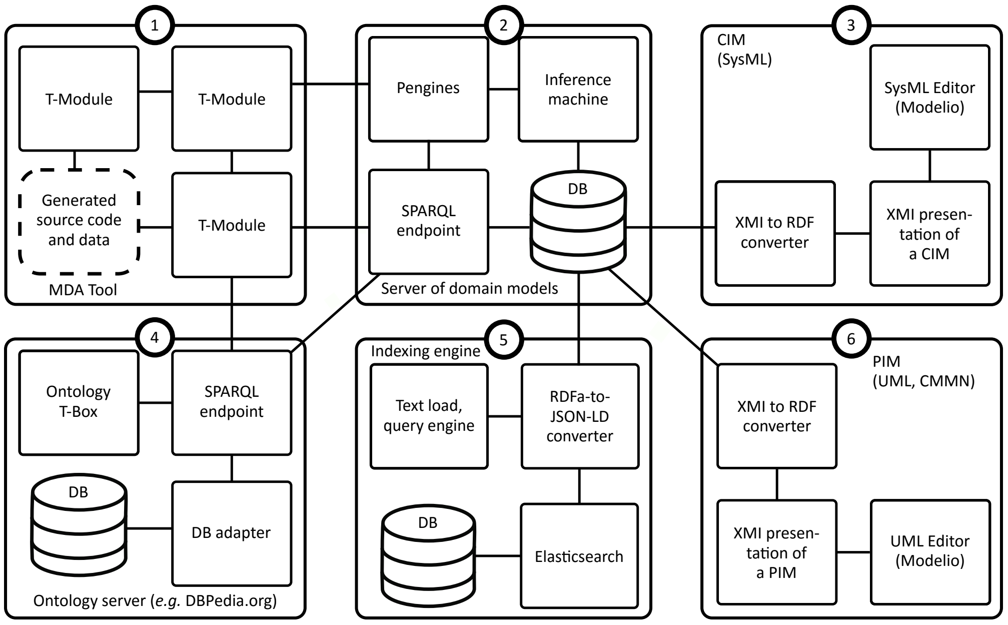

Fig. 1. Architecture of MDA tools

The MDA tools consists of the following main components:

- domain model server (2), which stores and retrieve model data as RDF triples;
- transformation MDA engine (1), which queries data from the domain server and auxiliary from other SPARQL compliant ontology servers (4);
- the main model data sources, representing CIM (3) and PIM (6);
- indexing engine (5), which provides persistent utility means for text data storage and text processing interfaces.

All components interact by means of network protocols and use utf-8 encoded HTML, TXT, XML and JSON formats for data representation, which are expressive enough and easily debugged. Inference engines also support SPARQL protocol for issuing subqueries.

The kernel of the system is MDA Tool, which is a set of transformational modules (T–Modules). Each of T–Module in a general case is a parameterized LogTalk object. The object queries the Server of domain models, which is built on the base of ClioPatria [2], other objects and other Ontology servers for structural patterns. These patterns are analyzed and the results are stored as the objects states, *i.e.* cached. The states of the object represent Platform Specific Model (PSM), which is translated into source code and initial data. Input structures of CIM and PIM models are converted into graphs with converter adapters. For example in Fig. 1 CIM (3) drawn in Modelio [3] is converted to triples with a special module "XMI to RDF converter". Similar converter is realized for PIM represented in UML. This approach makes the input stage modular, and, consequently, easier extensible with new model types.

There are number of partially structured data sources related to terms reflecting common objects. For example, `DBPedia.org` contains naming in various languages, as well as relations, of the entities represented in Wikipedia. This information can be used to describe user interface properties of the program objects, and referred by URI's from devised models. For example, we can use DBPedia for constructing `title` and `placeholder` attributes of an input widget, together with its label name text in user's locale and a help description.

3 RDF representation of the models

The past 20 years of unhurried Semantic Web (SW) technologies development formed descriptions of commonly used domains and represented them as standard W3C approved ontologies. It enables system designer and programmer to use globally identified resource references (URI), and ontology representation standards (RDF, RDFa, TTL, etc.), which are implemented in various programming environments as libraries. This results in a globally approved (in various software projects) knowledge and data base, the common foundation of a domain description. This is why, we decided to represent the source model data in the SW framework too, efficiently enriching the existing knowledge with a specific domains. From other hand side, converting the complex tree-like structures used for model representation to set of triples, i.e. facts for inference machine, enables us to use Prolog based engines for transformation definition.

In order to meet SW requirements, each model should be identified with a global URI, and in addition, its prefix could be defined. All relations and attributes must be expressed in a SW ontology, which should preferably be a standard one and have a publicly accessible formal definition. The entity names and their relation expressed with the source notation must be convertible to the RDF definition upon request from transformation engine. [[what else must be said about the representation, e.g. details of the transformation engine...]]

3.1 Entity denotation

Entity identification and naming in RDF representation have various aspects. An entity (or a relation) exists if it is defined or referred with its URI. In addition, an entity can be named, i.e. for its

URI is defined a naming relation, e.g., `dc:title` or `rdfs:label`. The relation `rdfs:range` refers to a `literal` string with a corresponding name in a natural language (defined by an attribute). So, each structure object or a relation in model notation must have a corresponding notation in an model representation ontology.

As the entities have global identification, their mentioning in different models denote the same object, e.g. in a BPMN diagram, a record object could be created of a type defined in an UML Class Diagram. In this case the type is double referenced object of the software model. The references of an entity of a model from another model can be done as RDF names of a form `<prefix>:<identifier>`, where `<prefix>` is the prefix of the model, where `<identifier>` is defined. Most UML and other modeling software do not restrict names of the identifiers, the restrictions are applied with a transformation tools.

[[What else it is necessary to describe here?]]

3.2 Model extensions

A flexible modeling tools also allow designer to extend a notation of a model (metamodel). For example, all UML diagrams support *stereotypes*, *tag values*, and *object constrains*. Firstly, stereotypes denoted a kind of a described object (it can be assigned to any entity or relation), e.g., in a Class Diagram stereotypes is used to define the kind of a class. If a class box is assigned standard `<<interface>>` stereotype then it is an interface definition. By default all classes are standard `<<class>>`-es. The same is used to describe relations between use cases in Use Case diagram. A use case can `<<include>>` another one or be `<<extend>>`-ed with another use case. User-defined stereotypes can express various aspects of classes, e.g., we can define the storage class as an ORM[4] of a class by `<<RDBMSRecord>>` stereotype. In this case, class is added to set of classes describing a relation database records, all relations between such classes are interpreted as rational (one–to–one, one–to–many, many–to–many, mandatory, etc.) relations. One entity can be assigned many stereotypes.

Tag values allow one to associate a string value identified by a string key, which are somehow interpreted with a traditional procedure. For example, the easy way of providing of an interface label name for an object `age` field in a class defined in a Class Diagram is an association the field with tag name `interface-label-name` as `''Age''` string. Using RDF, we can change the tag name to `dc:title` and value can be interpreted as URI, e.g. to a `DBPedia` object.

The usage experience of stereotypes and tag values resulted in recognition of the fact, that assigning a stereotype usually implies definition of the same set of tag values. UML standard of version 2.4 fixed this observation, allowing formal definition the set of tag values for user-defined stereotypes. Special user interfaces were developed to control assignment a stereotype to an entity. As stereotype does not provide a value in its association, table name for `<<RDBMSRecord>>` an optional tag value can be defined for this purpose, e.g., `rdbms:table-name`. UML-2.4 explicit stereotype descriptions is a powerful tool for attributing entities with additional information.

[[[Resume of some kind]]]

4 Related work on ontology usage in MDA

SysML is the dedicated system-level UML-based notation proposed by the OMG. In research [4] system design productivity is addressed as one of the main challenges. Some suggested approaches are increasing the level of abstraction and automation, as well as producing executable specifications. In [5], model driven development is used for the construction of complex embedded systems integrating software and firmware. A SysML system model is devised according to the platform-based design paradigm, in which a functional model of the system is paired to a model of the execution platform. Subsystems are refined as Simulink models or hand-coded in C++.

[4] Object relation mapping

The SysML standard is attracting more attention of hardware designers as UML and SysML have been used to automatically generate an HDL code written in SystemC, Verilog and VHDL. In [6], contrarily to the existing works, authors propose the new reverse engineering approach to generate SysML definition of block and internal block diagrams from VHDL code. Code generation is done on the basis of a set of well-defined mapping rules between SysML and VHDL concepts.

UML profiles like SysML and MARTE have been a major research topic in electronic system design, but are mainly applied for specification and analysis in early design phases. Research [7] addresses the problem of the High-Level Synthesis (HLS), *i.e.* physical implementation aspect of electronic systems, which need diversity of design models and levels of abstraction. To overcome the conflict between a higher degree of abstraction and necessary details for further synthesis, modular interfaces are introduced as object-oriented synthesizable technique. In [7], SysML is used as an adequate modeling language for modular interfaces and C/C++/SystemC-based HLS. Authors extended SysML with annotations for synthesizable SystemC and high-level synthesis constraints and implemented a code generation scheme to achieve design flow automation. They used SysML editor Artisan Studio with industrial case study to demonstrate the applicability of SysML as a front-end for HLS.

In [8] UML/SysML is used to model avionics. The implementation language is C# with using libXML for XMI import and processing. Generated objects represent each syntactic element (`struct`, `enum`, function, *etc.*), which represents the target source code. Authors implemented only State Machine diagram transformation.

Some experience of representing document flows are obtained by commercial software vendors developing Directum, Documentum systems automatizing Russian municipal institutions and state corporations. Directum uses SysML state–like block diagrams to define flow of documents and the ISBL object-oriented language to express sophisticated behavior.

In our research we focus on representation of ISO 9001 entities in SysML and other DSLs translatable to RDF, and the MDA transformation logical inference representable as LogTalk objects.

5 SysML representation of QMS

QMS is represented in SysML with nine diagrams [9]. QMS requirements is represented with new (with respect to UML) *Requirement Diagram*, describing stated structural and logical constraints. Structural aspects are represented with following UML2 diagrams:

- *Block Definition Diagram* decomposes structure of activities, attributes and objects on elements;
- *Internal Block Diagram* models interaction of the decomposed elements and their data flow; its descendant new *Parametric Diagram* describes model of block attribute relationships;
- *Package Diagram* organizes models in packages in various ways.

Behavioral aspect of a QMS is expressed with UML2

- *Use Case Diagram*, which expresses functions of the system and their mutual relationships;
- *State Machine Diagram* defines the states of the blocks and transitions between the states;
- *Sequence Diagram* describes the order of messages between blocks and actors;
- *Activity Diagram* represents computational process of converting data enclosed in blocks.

BPMN2.0 diagram is used to describe imperative procedures, it is very expressive diagram representing functions and involving agents, together with their relationships. CMMN represents declarative aspects of the QMS especially data structures, processing stages, events and check points (milestones). Adding BPMN2.0 and CMMN diagrams to SysML diagram set creates a redundancy, but in the same time it allows one to use more rich tool set for QMS modeling.

6 Defining rules of transformation

Transformation process is organized as a scenario of connected transformational objects (in terms of LogTalk) [10]. Scenario is programmed as a special object defining transformation by means of calling `tr/N` rules. Rules recognize structures in the input graph elements and constructs code blocks.

```
:- object(direct(_Package,_LocalProf,_CodeProf)).
:- public([tr/4,tr/3]).
:- protected([package/1, profiles/2, profile/1]).
package(Package):- parameter(1, Package).
profile(Profile):- parameter(2, Profile).
profile(Profile):- parameter(3, Profile).
profiles(L):-
    findall(Profile, ::profile(Profile), L).
tr(class, Class, ClassID):- ::package(Package),
    query(Package)::class(Name, ClassID),
    create_object(Class,
        [instantiates(class)],[],[]),
    create_object(Attributes,
        [instantiates(params)],[],[]),
    create_object(Methods,
        [instantiates(methodlist)],[],[]),
    Class::name(Name),
    forall(
        ::tr(attribute,Attribute,ClassID,_AttrID),
        Attributes::append(Attribute) ),
    forall(
        ::tr(method, Method, ClassID, _MethodID),
        Methods::append(Method) ),
    Class::attributes(Attributes),
    Class::methods(Methods).
tr(attribute, Attribute, ClassID, AttributeID):-
    ::package(Package),
    query(Package)::attribute(Name,ClassID,AttrID),
    create_object(Attribute,
        [instantiates(param)],[],[]),
    Attribute::name(Name).
tr(method, Method, ClassID, MethodID):-
    ::package(Package),
    query(Package)::method(Name,ClassID,MethodID),
    create_object(Method,
    [instantiates(method)],[],[]),
    Method::name(Name).
:- end_object.
```

The previous listing uses static parameterized object `query/1`, whose argument is a graph representing model under transformation. This is a powerful LogTalk abstraction instrument, which reduces necessity of adapter object creation in dynamic memory. The SPARQL is encapsulated in methods.

```
:- object(query(_XMI)).
:- protected(xmi/1).
:- public([class/2, attribute/3, method/3]).
xmi(XMI) :- parameter(1, XMI).
class(Name, ID):- ::xmi(XMI),
    XMI::rdf(ID,rdf:type,uml,'Class'),
    XMI::rdf(ID,rdfs:label, literal(Name)).
attribute(Name, ClassID, ID):-  ::xmi(XMI),
    XMI::graph(G),
    XMI::rdf(ClassID, G:ownedAttribute, ID),
    % XMI::rdf(ID, rdf:type, uml,'Property'),
    XMI::rdf(ID, rdfs:label, literal(Name)).
method(Name, ClassID, ID):- ::xmi(XMI),
    XMI::graph(G),
    XMI::rdf(ClassID, G:ownedOperation, ID),
    XMI::rdf(ID, rdfs:label, literal(Name)).
:- end_object.
```

Procedure of source code generation is build using `code_block`–objects (the idea was taken from `llvmlite` library), which provide PSM representation:

```
:- object(code_block, specializes(root)).
:- public([append/1, prepend/1, clear/0,
   render/1, render_to/1, remove/1, item/1,
   items/1 ]).
:- dynamic([item_/1]). % elements of
:- private([item_/1]). % the code block
:- protected([renderitem/2, render_to/2]).

item(Item)   :- ::item_(Item).
items(Items) :- bagof(I, ::item(I), Items).
append(Item) :- ::assertz(item_(Item)).
prepend(Item):- ::asserta(item_(Item)).
remove(Item) :- ::retract(item_(Item)).
clear        :- ::retractall(item_(_)).
render(_)    :- writef::writef("ERROR: \
Implement render/1 by a subclass!\n"), fail.
render_to(Stream):- ::render(List),
    ::render_to(List, Stream).
render_to(List, Stream):-
    lists::is_list(List),!,
    forall(lists::member(X,List),
       ::render_to(X, Stream)).
render_to(X,_) :- write(X),nl.
renderitem(Object, String):-
    current_object(Object), !,
    Object::render(String).
renderitem(literal(Item), String):-!,
    atom_string(Item, String).
renderitem(Item, String):-
    root::iswritef(String, '%q', [Item]).
:- end_object.
```

The `code_block`–objects consist of source code string lines and other objects, including other code blocks. Each item is denoted with `item/1` and its private database element `item_/1`. The type of the element is defined by outer functor of the argument, *e.g.*, we define with `attributes(L)` a list `L` of attributes of a class. The way of rendering the sources is implemented by subclassing the object and realizing `render/1` and `renderitem/2`. The first argument of `renderitem/2` is the `item/1`–structure to be rendered, and the second one is the list of strings representing generated source code. Elements of the code block can be `append`ed, `prepend`ed and `remove`ed. We specially disallow to insert `items` into the database, as it will make object and its database more complex. Instead, programmer can add an `item` being another code block, inserting elements of the sources in it.

The following example shows rendering class for Python classes. It contains public methods for defining class elements.

```
:- object(class, specializes(code_block),
   imports([named])). % A category of named entities
:- public([classlist/1, methods/1, attributes/1]).
classlist(ClassList):- % List of base classes
    ::prepend(classlist(ClassList)).
attributes(Attributes):- % List of attributes
    ::prepend(attributes(Attributes)).
methods(MethodList):- % List of methods
    ::append(methods(MethodList)).

renderitem(Item, Result):- % Default
    ^^renderitem(Item, Result).
render(Result):- % Render the class
    ^^render(Name),
    ( ::item(classlist(List)) ->
      List::render(ClassList),
      root::iswritef(Signature,'class %w(%w):',
        [Name, ClassList]);
      root::iswritef(Signature,'class %w:',
        [Name]) ),
    root::indent, % add an indent
    ( ::item(attributes(Attributes))->
      Attributes::render(DefAttrList),
      root::iswritef(ConstructorDef,
        'def __init__(self, %w):',
        [DefAttrList]),
      root::indent,  % more indent
      Attributes::items(InstanceAttrs),
      findall(S, ( % initialize attributes
         lists::member(Attr, InstanceAttrs),
         Attr::item(name(AttrName)),
         root::iswritef(S, "self.%w=%w",
           [AttrName, AttrName])
         ), AttrAssigns),
        root::unindent,
        AttrList=[ConstructorDef|AttrAssigns];
        root::iswritef(ConstructorDef,
          'def __init__(self): ', []),
        root::indent,
        root::iswritef(Pass,'pass', []),
        root::unindent,
        AttrList=[ConstructorDef, Pass] ),
    ( ::item(methods(Methods))-> % if any ...
      Methods::render(MethodList);
      MethodList=[] ),
    lists::append(AttrList,MethodList,StringList),
    root::unindent, Result=[Signature|StringList].
:- end_object.
```

In the same way, we can construct procedures for creating structures of databases and populating them with initial data.

7 Conclusion

The idea of formal description of the business process during transition of a medical institution for compliance to ISO 9001:2015 standard is considered in the paper. Diagrams SysML, BPMN2.0, CMMN are proposed to be used for this purpose. The description provides Computational Independent Model (CIM) of Model Driven Architecture (MDA) software development approach, and it is to be transformed in subsystems of Informational System (IS).

CIM is being designed with visual editors, *e.g.* Modelio [3], translated from their XMI to RDF and stored as ontology graphs as files or network resources. MDA-transformation is represented and executed within Prolog programming environment LogTalk [10], which provides us knowledge structuring, rich set of libraries and uniform well-known programming language for defining transformation rules. The general scheme of transformation definition is presented.

IS development tools is being tested in representation of bioinformatic tool Mothur as Rapidminer dataflow diagrams [**?**]. The source data for transformation is the module specifications and UML Class Diagram, which organizes submodules in virtual class hierarchies.

8 Acknowledgments

The results are obtained with the partial support of Russian Foundation for Basic Research, grants 18-07-00758. The results obtained with the use of the network infrastructure of Telecommunication center of collective use “Integrated information-computational network of Irkutsk scientific-educational complex” (`http://net.icc.ru`).

References

1. [[[A book about technical audit.]]]
2. *Wielemaker,J.*, *Beek,W.*, *Hildebrand,M.*, *Ossenbruggen,J.* ClioPatria: A SWI-Prolog Infrastructure for the Semantic Web, Semantic Web. vol. 7, no. 5, 2016, pp. 529-541. DOI:`10.3233/SW-150191`
3. Modelio Open Source – UML and BPMN free modeling tool. URL:`https://www.modelio.org/`.
4. *Raslan, W.*, *Sameh, A.* Accelerating High-Level SysML and SystemC SoC Designs, URL:`https://www.design-reuse.com/articles/17562/high-level-sysml-systemc-soc-designs.html`
5. *Natale,M.*, *Perillo,D.*, *Chirico,F.*, *Sindico,A.*, *Sangiovanni-Vincentelli,A.* A Model-based approach for the synthesis of software to firmware adapters for use with automatically generated components, Software & Systems Modeling, February 2018, Volume 17, Issue 1, pp. 11-–33
6. *Boutekkouk,F.*, *Fartas,O.* Automatic generation of SysML diagrams from VHDL code, In Proceedings of the Symposium on Complex Systems and Intelligent Computing (CompSIC), Souk Ahras, Algeria, September 2015.
7. *Mischkalla,F.*, *He,D.*, *Mueller,W.*, *Azcarate,F.*, *Carballeda,M.* A Retargetable SysML-based Front-End for High-Level Synthesis, In: Proceedings of 2nd Workshop on Model Based Engineering for Embedded Systems Design (M-BED), Mrz. 2011
8. *Hauswald,T.* Automatic Code Synthesis of UML/SysML State Machines for airborne Applications. Bachelor Thesis. Hamburg Univeristy of Technology. August 15. 2016. 80 p.
9. *Friedenthal,S.*, *Moore,A.*, *Steiner,R.* OMG Systems Modeling Language (OMG SysML™) Tutorial. 2009. 132 p. `http://www.omgsysml.org/INCOSE-OMGSysML-Tutorial-Final-090901.pdf`
10. *Cherkashin,E.*, *Larionov,A. et al.* Logical programming and data mining as engine for MDA model transformation implementation. Procs of 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). May 20–24, 2013, Opatija, Croatia, pp. 1029–1036.
11. Digital Archives
12. ICIST