Representation of MDA Transformation with Logical Objects

Evgeny Cherkashin, Alexey Shigarov, Viacheslav Paramonov
Matrosov Institute for System Dynamics and Control Theory of SB RAS,
Irkutsk Scientific Center of SB RAS
134 Lermontov Street, Irkutsk, 664033, Russia
{eugeneai,shig,slv}@icc.ru

***Abstract*—The problem of software modeling having various models as sources and its transformation based on logical inference is considered. Source models are converted into graphs of RDF and then processed with knowledge based system organized in a network of objects. Various notation can be used to represent models, such as UML, SysML, CMMN, BPMN2.0, as well as RDF graphs and analyzed source code, having implemented a corresponding converter. The objects are represented in the LogTalk programming language. Objects query graphs and other objects implementing a scenario of a software system synthesis within Model Driven Architecture paradigm.**

Usage of such kind of transformation approach allows us to develop software system carcasses on the level of abstract models, involve various sources of model data in the transformation, define and structuring conversion knowledge as objects. An example of a dataflow environment synthesis encapsulating Mothur library for new generation sequencing is presented.

***Index Terms*—model driven architecture, logical inference, resource description framework, LogTalk**

I. Introduction

One of the main challenge of informational systems (IS), as well as other industrial software design, development is complexity expressed in its structure, integration with other software, fast prototyping on the stage of design. Rising the stage of code generation of IS within the development to abstract level of design allows us to produce IS prototypes as early as the specification are formalized. CASE–systems are usually developed close to standard industrial technologies; it is not popular in developing IS for small organization and startups, which is characterized with uncertainties of requirements and specification.

In order to support IS subsystems generation, we are to develop an expressive means for transformation from set of formalized models represented in UML, SysML, BPMN and other ones constructed with corresponding design tools. The process of tradition programming should allow us to develop specialized versions for peculiar software design approaches, with conserving common practices as libraries. One of the paradigm of the implementation could be object–orientation, which in context of knowledge based systems will allow us to have a powerful tools enabling knowledge structuring, manipulation, and encapsulation as transformation components.

The most investigated transformational approach of code generation, that goes beyond the CASE, is Model Driven Architecture (MDA). MDA represents code generation as multistage process of model transformation. The source code and initial data are produced from a Platform Specific Model (PSM) that represents IS implementation on a specific program and hardware platform. PSM is obtained from a Platform Independent Model (PIM), which describe IS more abstract without platform specifics. A part of PIM could be resulted from a Computational Independent Model representing the IS and its environment on conceptual, organizational, system level, which does not define information conversion (e.g., computation).

CIM and PIM usually defined as a sets of SysML, CMMN, UML and other visual models, which saved in a standard way as XMI[1] files. Dispute these notations additional data for transformation can be obtained from Semantic Web (SW), for example, ontology definitions, literal names of entities in various natural languages, current state of processes, and so on. The popular ways of model transformation, like ATL[2], are based on transformation one XMI model to another XMI, and then to the code. Involving SW data is problematic.

We propose a more general approach, where for all model sources a converter to RDF[3] format, the standard SW data and knowledge representation, and the transformation is defined as object-oriented logical program in the LogTalk language. This allows us represent the transformation as a scenario of objects exchanging messages, querying a SW graph fact database on model structures and SW data. Object programming enables us to structure, manipulate the knowledge base with encapsulation, inheritance, extending and composition.

The *object* of the research is the development of a technology for transformation implementation as knowledge based system and be capable to process heterogeneous model sources as a complex of CIM and PIM. The *subject* of the paper is to describe above proposed data representation and transformation techniques.

II. Architecture of MDA tools

The architecture of the transformation 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 SW graphs of

[1]XML Metadata Interchange
[2]ATLAS Transformation Language
[3]Resource Description Framework

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.

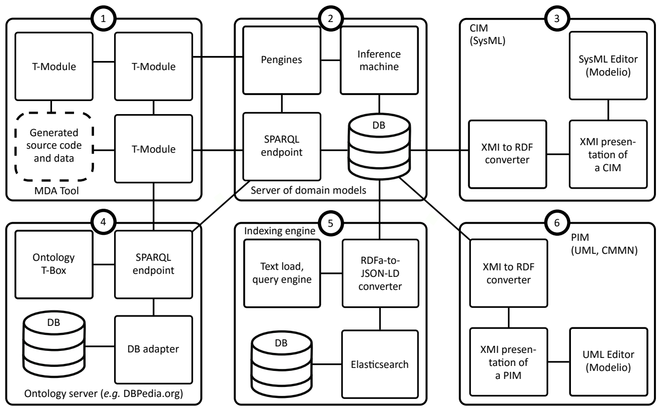

Fig. 1. Architecture of MDA transformational tools

The MDA tools consists of the following main components:

- source model server (2), which stores and retrieve model data as RDF triples;
- MDA transformation engine (1), which queries data from the server other SPARQL compliant servers (4);
- the CIM (3) and PIM (6) model data sources;
- 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 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 [1], other objects and other Ontology servers for structural patterns. These patterns are analyzed and the results are stored as the objects states, *e.g.*, cashed in the indexing engine. The states of the object represent 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 [2] 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.

One of auxiliary RDF–compliant data sources is `DBPedia.org` containing naming literals 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 a web input widget, together with its label name text in user's locale and a help description.

III. RDF representation of the models

The past 20 years of unhurried SW technologies development formed descriptions of commonly used domains and represented them as 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, 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.

A. Entity denotation

Entity identification and naming in RDF have various aspects. An entity or a relation exist if they are defined or referred with its URI. An entity can be named by means of a relation with its URI, *e.g.*, `dc:title` or `rdfs:label`. The relation refers to a `literal` string with a corresponding name in a natural language (defined by an attribute). Each model structure element or a relation in a 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. 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, together with their semantics, which is defined with transformation tools.

B. 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*. Stereotypes declare an extended set of properties for an object or relation, *e.g.*, in UML Class Diagram stereotypes are used to define the kind of a class. If a class 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] for 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 transformation 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 a set of regular 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.

C. Other sources of model data

When implementing complex software, which are composed from existing subsystems, their specifications can be additional sources of model data. For example, in our project [8], the main PIM model is extracted from source code of the original library Mothur and converted to TTL (turtle) representation. This approach requires one-time programming for adaptation of the sources, but for each new version of the library we can obtain new version of the PSM without additional analysis.

IV. Related works

The most widely used technology of model transformation is ATL (ATLAS Transformation Language) [9] and its predecessor QVT, an OMG standard [10]. The language and its engine supports conversion from one XMI model to another one in the same XMI format. The language structures describe recognition of properties of compositions of the source model and direct construction of new structures in the target model.

Similar to ATL, Transformation Model Representation Language (TMRL) is developed in [11] but oriented on visual representation of a knowledge base with following conversion in various production rules in CLIPS and OWL. The visual presentation of transforming rules are used also in [3]. In [4], ATL is used to transform Computational Independent Model (CIM) represented as BPMN diagram into set of UML diagrams, defining PIM. Paper [5] proposes a model transformation CIM to PIM for web-applications, where CIM is represented with State and Use case UML-Diagrams logically connected with ATL rules. In [6] is proposed an evaluation of security aspects of distributed applications using MDA description; the evaluation is implemented a logical inference over the MDA synthesized logical security models. The paper [7] is considered a highly decoupled distributed event-driven application environment, whose design and functioning is dynamically controlled by its three-level DSL[5] description and MDA transformations, which support change propagation.

Thus, the majority of the presented techniques are closed with respect of XMI file format: the source data and the result of the transaction are represented in it, and it goes beyond only on the stage of the source code generation. Our approach allows one to use other sources (not being presented in XMI) of model information, as well as general purpose libraries in implementation of the transformation engine. Usage of object-oriented logical language LogTalk allows us powerfully structuring and manipulate knowledge base.

V. Technique of Transformation Implementation

The main idea behind implementation of transformation engine is an usage of an object logical programming language meeting the following constraints:

- support multiple sources of model data,
- first-order language for definition of the transformation,
- transformation is represented as rules,
- rules should be encapsulated in objects,
- objects should query other objects for results,
- means for transformation configuration and scenario description,
- knowledge manipulation, including composition,
- expression and query of RDF entities.

SWI-Prolog with macro package LogTalk meet these constraints. The LogTalk objects encapsulate Prolog first-order language rules as methods. Each object is respected as a knowledge base of encapsulated rules. Objects can be inherited (as prototypes or classes), parameterized, and composed with *categories*. In inheritance, methods can be reprogrammed implementing refined knowledge set. Configurations are implemented as a prototyping hierarchies: in inheritance, a configuration are refined by replacing and adding new values. Scenarios are implemented as an ordered sets of methods. Finally, SWI-Prolog has a good LogTalk compatible RDF library strongly tied to the language itself. The library represents RDF entities in a W3C compliant way.

A. SWI-Prolog Semantic Web Library

SWI Prolog standard distribution contains library for RDF processing. The library represents URI as a corresponding

[4]Object Relation Mapping

[5]Domain Specific Language

functor with atom argument. This structure can be constructed with a special form `<namespace>:<identifier>`. Library contains predicates for loading graphs from files and internet sites, store them in various text based and binary formats for later use.

The library support queries for the graphs as a predicate query and as SPARQL. There is a basic query optimization. For the subclass and subpropery relation, library realizes transitive closure, which can be used in the predicate queries. The library imported in LogTalk as an object, that is an encapsulation of all the library predicates.

B. LogTalk Methods

LogTalk object consists of only encapsulated predicates (methods), some of them can be defined as dynamic, implementing object state. LogTalk as a macro package provides two kind of objects: static and dynamic. Static objects is defined and instantiated during compilation, dynamic ones are created with `create_object/4` usually at runtime. Static parameterized objects can be considered as third way of instantiation. The role of an object is defined with relation to other objects, *e.g.*, instances of a class are the objects, which are `instantiates/1` its class.

Static object instances are used for defining configurations, scenarios, interfaces to input and output targets (streams, models), graph databases, and other global compile time known entities. Their main role is structuring Prolog rule set. Dynamic objects are used in construction of PSM objects, currently, each instantiation is an entity of target programming language. Parametrized objects are facade interfaces to other objects (contexts), *e.g.*, input targets and graph databases. These objects enables programmer to define recognition rules over set of structures provided it with parameters. Parameterized objects should be stateless, in this case, LogTalk compiles it in statically executed code.

C. PSM Representation

PSM and source code generation are represented with `code_block` objects. The idea was taken from `llvmlite` library. The code block object have the following interface:

```
:- protocol(code_block_proto).
:- public([
   append/1, prepend/1,  % Manipulation of PSM
   clear/0,              % content as a list
   remove/1,             % of ordered items.
   item/1, items/1]).    % List the items.
:- protected([
   renderitem/2,  % Converting items to code lines.
   render_to/2]). % Storing lines to a stream.
:- end_protocol.
```

The `code_block`–objects consist of source code string lines, structures denoting source code, objects, and other code blocks. Each item is accessed via `item/1`. The type of the element is defined by outer functor of the argument, *e.g.*, we define with `append(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 a code block can be `append`ed, `prepend`ed and `remove`ed. We specially disallow inserting `item`s into code blocks, as it will make object and its database more complex. Instead, programmer can add an `item` being another code block, inserting its content in the outer block.

If needed the item's definition can include provision information which can be used in result construction, when its object being queried.

D. Defining rules of transformation

Transformation process is organized as a scenario of connected transformational objects (in terms of LogTalk). 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,  % inherits code_block
        [instantiates(class)],[],[]),
    create_object(Attributes,
        [instantiates(params)],[],[]),
    create_object(Methods,
        [instantiates(methodlist)],[],[]),
    Class::append(name(Name)),
    forall(
        ::tr(attribute,Attribute,ClassID,_AttrID),
        Attributes::append(Attribute) ),
    forall(
        ::tr(method, Method, ClassID, _MethodID),
        Methods::append(Method) ),
    Class::append(attributes(Attributes)),
    Class::append(methods(Methods)).
tr(attribute, Attribute, ClassID, AttributeID):-
    ::package(Package),
    query(Package)::attribute(Name,ClassID,AttrID),
    create_object(Attribute,
        [instantiates(param)],[],[]),
    Attribute::append(name(Name)).
tr(method, Method, ClassID, MethodID):-
    ::package(Package),
    query(Package)::method(Name,ClassID,MethodID),
    create_object(Method,
       [instantiates(method)],[],[]),
    Method::append(name(Name)).
:- end_object.
```

The previous listing uses static parameterized object `query/1`, whose context 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 queries are 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.
```

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), % Call inherited method.
    ( ::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.
```

Let us do some remarks. Class method `root::iswritef/3` is used to generate source code with a tab indent organized by `root::indent` and `root::unindent` predicates. List `Result` of `render/1` specially not flatten in order to not use append operations. The object imports category `named`, which defines behavior for named instances.

LogTalk *categories* is very useful way of implementing a common functionality among a set of entities. For example, programming language objects have an identifier and, optionally, type description. These properties are implemented as two categories. `named`, and its child `namedtyped`.

```
:- category(named).
:- public([name/1, render/1]).
:- protected([renderitem/2]).

name(Name):- ::prepend(name(Name)).

renderitem(name(Name), String):-!,
    atom_string(Name, String).

render(String):-
    ::item(name(Name)),
    ::renderitem(name(Name), String).

:-end_category.
```

The category implies interface of `code_object` for the object being enriched with it. The second category definition as well as all the sources of transformation objects are stored at `github.com` [12].

VI. Application in NGS

The proposed technique is applied in developing software for visual scripting NGS[6] computational procedures such as sequencing data loading, filtering, gene alignment, result assessment [8]. The procedure is represented as dataflow diagram constructed and evaluated in Rapidminer system. Each block of the diagram defines an operation, which is carried on over a set of input files by one of 147 Mothur modules; the result is stored in output files. The files are transferred between modules. This is reflected as connections between blocks in the diagram. The visual instrument is intended for biologists allowing them to make bioinformatic exploration themselves.

Each block structure, namely, the sets of input and output connections, together with parameters, reflects the interface of Mothur modules. The modules are implemented as classes in C++ with special structures describing each component so as this description would be allowed for analysis at run-time. We use this structural data of the source code of Mothur for analysis and conversion it to a RDF graph, resulting in the description of all the Mothur modules.

The analysis of the sources is performed with regular expressions and simple translation units. We tried to use a version of GCC[7], which outputs abstract syntax tree as an XML file. Unfortunately, the structure of this file is too

[6]New Generation Sequencing
[7]GNU C Compiler

complex to analyze and contained only declarative parts of the sources (types, classes with field and method declarations and external variables).

Transformation generates Java modules implementing structural and operational features of blocks. The logic of some Mothur modules functioning requires to make flexible set of output connection to make usage of complex modules to be more convenient for end user. Now we are working on extending the knowledge base to generate algorithms figuring out the set of acceptable connections depending input combination and module configuration.

Overall time spent to develop the proposed transformation technique and the basic (structural) aspects of Mothur module representation is two human-months. The implemented internal logic allowed us to determine some errors in Mothur source module descriptions. So the transformation modules can be used as a verification system for the sources.

One of the system's functioning mode will be interaction with big data storage, so after refining in the current stand–alone mode we are to develop a variant of the visual tools, which would able to query needed content from the storage and generate LOD[8]–marked up reports. LOD is based on RDF so our input model representation data can be used in generation the markup of the report templates and algorithms.

VII. Conclusion

We presented an approach to implementation of transformation procedures for Model Driven Architecture (MDA) that mediates most convenient features of logical and object-oriented programming, representation of source model data as RDF graphs. The usage of LogTalk language allowed us to use object facilities to structure transformational knowledge, create an instrumental basis for manipulation knowledge sets with object inheritance and composition. The encapsulation gave us possibility to hide specialized knowledge behind object interfaces.

The development tools is being tested in representation of bioinformatic tool Mothur as Rapidminer dataflow diagrams [8]. The source data for transformation is the module specifications and UML Class Diagram, which organizes submodules in virtual class hierarchies. The result is a Rapidminer plug-in, with each Mothur module being represented as a data block.

With these MDA tools now we will develop a programming techniques aimed at reuse of knowledge and their specialization, thus, supporting programming traditions used in small information system developing companies and collectives. Another aspect of research is integration with existing Linked Open Data sources such as [13].

VIII. Acknowledgements

The presented research results is distributed among three grants as follows: aspect of LOD data representation and integration is supported by Russian Science Foundation, grant 18–07–0075; logical inference combining various source models and object–oriented knowledge representation is supported by Russian Foundation of Basic Research, grant 18–71–10001; the presented example of Rapidminer dataflow modules generation is being developed with support of grant of Irkutsk Scientific Center SB RAS, project No 4.2.

[8]Linked Open Data

References

[1] J. Wielemaker, W. Beek, M. Hildebrand, J. Ossenbruggen, "ClioPatria: a SWI-Prolog infrastructure for the Semantic Web," Semantic Web. vol. 7, no. 5, 2016, pp. 529-541.

[2] "Modelio open source – UML and BPMN free modeling tool." URL:https://www.modelio.org/.

[3] A. Belghiat, M. Bourahla, "UML Class Diagrams to OWL ontologies: a graph transformation based approach," International Journal of Computer Applications. no. 41. pp. 41–46.

[4] Y. Rhazali, Y. Hadi, A. Mouloudi. "Model transformation with ATL into MDA from CIM to PIM structured through MVC,"Procedia Computer Science 83 (2016) 1096-–1101. URL:https://doi.org/10.1016/j.procs.2016.04.229

[5] Y. Rhazali, Y. Hadi, I. Chana, M. Lahmer, A. Rhattoy, "A model transformation in model driven architecture from business model to web model," IAENG International Journal of Computer Science, vol. 45(1), 2018, 104-117, URL:http://www.iaeng.org/IJCS/issues_v45/issue_1/IJCS_45_1_16.pdf.

[6] B. Hamid, D. Weber, "Engineering secure systems: models, patterns and empirical validation," Computers & Security, vol. 77, 2018, p. 315-348.

[7] S. Tragatschnig, S. Stevanetic, U. Zdun, "Supporting the evolution of event-driven service-oriented architectures using change patterns." Information and Software Technology. vol. 100, 2018, p. 133-146.

[8] E. Cherkashin, A. Shigarov, F. Malkov, A. Morozov, "An instrumental environment for metagenomic analysis," In: Bychkov I., Voronin V. (eds) Information Technologies in the Research of Biodiversity. Springer Proceedings in Earth and Environmental Sciences. Springer, Cham, 2019, pp. 151–158.

[9] F. Jouault, F. Allilaire, J. Bezivin, I. Kurtev, "ATL: a model transformation tool," Sci. Comput. Program. vol. 72(1–2), pp. 31–39 (2008)

[10] "The MOF query/view/transformation specification version 1.1." URL:http://www.omg.org/spec/QVT/1.1

[11] A. Berman, M. Grishchenko, N. Dorodnykh, O. Nikolaychuk, A. Yurin. "A model-driven approach and a tool to support creation of rule-based expert systems for industrial safety expertise," Proc. of the 12-th International Forum on Knowledge Asset Dynamics (IFKAD-2017) – Russia, St. Petersburg : Graduate School of 16 Management of St. Petersburg University. 2017. P. 2034–2050.

[12] E. Cherkashin, "GitHub project page of ICC.XMITransform project." URL:https://github.com/isu-enterprise/icc.xmitransform (access date: 01-oct-2019)

[13] E. Cherkashin, A. Shigarov, V. Paramonov, A. Mikhailov, "Digital archives supporting document content inference," Procs of 42-nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 20–24 May, 2019, Opatija, Croatia. pp. 1037–1042.