# AN APPROACH FOR DESIGN OF ARCHITECTURAL SOLUTIONS BASED ON SOFTWARE MODEL-TO-MODEL TRANSFORMATION

**Elena Chebanyuk, Kyryl Shestakov**

***Abstract**: Software models are central software development artifacts in Model-Driven Software Engineering (MDSE) approach. During the Requirements Analysis, Software Design and Software Development a variety of software models, represented as UML diagrams, are created. However, in a majority of software development approaches they are designed mainly. Considering the fact that new software models contain information from previous ones, software Model-to-Model transformation techniques facilitate the process of new software models obtaining by means of reusing information from software models created before.*

*An approach for performing Model-to-Model transformation, which is based on graph transformation, is presented in this paper. Transformational operations are considered on meta-level and concrete level. Software models are represented as graphs.*

*Software tool, that allows performing transformation from communication diagram to class one, is described. This tool is designed as a plug-in for IBM Rational Software Architect (RSA) 7.5.5.2. Engine medini QVT, integrated in IBM RSA project, supports execution of QVT-R script. Source software models (communication diagrams) are designed in IBM Rational Software Architect. Target software models are also processed (opened) by means of IBM RSA.*

***Keywords**: Model-Driven Software Engineering, Model-Driven Development, Model-to-Model Transformation, Graph Transformation, Transformation Rule, IBM Rational Software Architect, medini QVT, Requirement Analysis, Software Designing, UML Diagram, UML Communication Diagram, UML Class Diagram.*

***ACM Keywords**:*

***- Classification of 2012 year**: Software and its engineering, Software system structure, Software system model, Model-Driven system engineering.*

***- Classification of 1998 year:** D2 software engineering, D 2.0 Tools*

## Introduction

Software modeling is an integral activity that is performed on software project development that follows a Model-Driven Development (MDD) approach. It is applied to express software with the aim of reducing risk and managing complexity of activities, performed in different software development processes. And the challenge of projects that heavily rely on modeling is to transform designed software models. A software model transformation, which is an automated way of processing models, can be written in a general-purpose programming language, such as Java. However, using a special-purpose transformation language is a more flexible approach since it does not require complex and redundant coding constructs but focuses on relevant domain features. Also using a special-purpose model transformation language increases maintainability, because changes introduced into transformation scripts require less effort to be implemented and no recompilation since scripts are not embedded into transformation software.

Previously, an approach to transform models into text was used exclusively because a final goal of the transformation was to obtain code for the development of software. Plus, the transformation was fixed and completely not capable of change in the implementation of a transformation tool. With the introduction of Model-to-Model transformation activity the objective did not change but rather an additional layer of abstraction was included in order to increase the quality of desired result. This is accomplished by taking advantage of existing semantic relations between different representations of a model to be developed.

The techniques employed for modeling are standardized by the Object Management Group (OMG). Among them are Unified Modeling Language (UML) to define software models expressed as diagrams, Object Constraint Language (OCL) to specify constraints to objects, etc. These widely-accepted techniques are often used to define software requirements in an Agile approach when interacting with a customer. After the meetings with a customer a great number of diagrams is created through which a model is generated. This serves as a substantial input for the process of Model-to-Model transformation. The key in the proposed approach is to transform into a model that reflects software architecture better than other representations. So Model-to-Model transformation fits mostly between the stages of software analysis and design.

In order to implement Model-to-Model software transformation process on a project that follows MDD approach the hardest part is to figure out appropriate relationships between the model elements of given and desired model views. This might make a lot of time analyzing, designing and developing software to gain insight on the relevant model patterns that can be mapped into each other under conditions that should hold. But according to Model-to-Model transformation approach, this kind of experience helps automate the process of software design, which reduces time and effort needed to develop software.

Instead of creating necessary design all the time wasting effort that could be invested into other pressing project matters, required software model can be generated from existing model.

This approach is customary to use at large projects that develop software with high level of complexity, because only such types of software can provide a necessary amount of models for the discovery and analysis of transformation rules. And due to their size, they can be primary beneficiaries of considered approach for saving time and effort, which are valuable resources that have direct correlation with money.

## Review of software Model-to-Model transformation types

Model transformations and languages for them have been classified in many ways. Some of the more common distinctions drawn are:

*Number and type of inputs and outputs:*

In principle a model transformation may have many inputs and outputs of various types; the only absolute limitation is that a model transformation will take at least one model as input. However, a model transformation that did not produce any model as output would more commonly be called a model analysis or model query.

*Endogenous versus exogenous:*

Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages. For example, in a process conforming to the OMG Model-Driven Architecture, a platform-independent model might be transformed into a platform-specific model by an exogenous model transformation.

*Unidirectional versus bidirectional:*

A unidirectional model transformation has only one mode of execution: that is, it always takes the same type of input and produces the same type of output. Unidirectional model transformations are useful in compilation-like situations, where any output model is read-only. The relevant notion of consistency is then very simple: the input model is consistent with the model that the transformation would produce as output, only.

For a bidirectional model transformation, the same type of model can sometimes be input and other times be output. Bidirectional transformations are necessary in situations where people are working on more than one model and the models must be kept consistent. Then a change to either model might necessitate a change to the other, in order to maintain consistency between the models. Because each model can incorporate information which is not reflected in the other, there may be many models which are consistent with a given model.

*Horizontal vs Vertical Transformation:*

A horizontal transformation is a transformation where the source and target models reside at the same abstraction level (e.g. Platform independent or platform specific levels). Typical examples are refactoring (an endogenous transformation) and migration (an exogenous transformation). A vertical transformation is a transformation where the source and target models reside at different abstraction levels. A typical example is refinement, where a specification is gradually refined into a full-fledged implementation, by means of successive refinement steps that add more concrete details.

*Syntactic versus semantic transformations:*

A final distinction can be made between model transformations that merely transform the syntax, and more sophisticated transformations that also take the semantics of the model into account. As an example of syntactical transformation, consider a parser that transforms the concrete syntax of a program (resp. model) in some programming (resp. modeling language) into an abstract syntax. The abstract syntax is then used as the internal representation of the program (resp. model) on which more complex semantic transformations (e.g. refactoring or optimization) can be applied. Also when we want to import or export our models in a specific format, a syntactical transformation is needed [Errata et al, 2015].

## Model Transformation Languages and Tools

### Atlas Transformation Language

Atlas Transformation Language (ATL) is a model transformation language and toolkit developed and maintained by OBEO and INRIA-AtlanMod. It was initiated by the AtlanMod team (previously called ATLAS Group). In the field of Model-Driven Engineering, ATL provides ways to produce a set of target models from a set of source models. Released under the terms of the Eclipse Public License, ATL is an Model-to-Model (Eclipse) component, inside of the Eclipse Modelling Project.

ATL is based on the QVT which is an Object Management Group standard for performing model transformations. It can be used to do syntactic or semantic translation. ATL is built on top of a model transformation Virtual Machine [Errata et al, 2015].

### Janus Transformation Language

Janus Transformation Language (JTL) is a bidirectional model transformation language specifically designed to support non-bijective transformations and change propagation. In Model-Driven Engineering, bidirectional transformations are considered as core ingredients for managing both the consistency and synchronization of two or more related models. However, while non-bijectivity in bidirectional transformations is considered relevant, most of the languages lack of a common understanding of its semantic implications hampering their applicability in practice.

The JTL is a bidirectional model transformation language specifically designed to support non-bijective transformations and change propagation. In particular, the language propagates changes occurring in a model to one or more related models according to the specified transformation regardless of the transformation direction. Additionally, whenever manual modifications let a model be non-reachable anymore by a transformation, the closest model which approximate the ideal source one is inferred. The language semantics is also presented and its expressivity and applicability are validated against a reference benchmark. JTL is embedded in a framework available on the Eclipse platform which aims to facilitate the use of the approach, especially in the definition of model transformations [Cicchetti et al, 2010].

Epsilon family is a model management platform that provides transformation languages for model-to-model, model-to-text, update-in-place, migration and model merging transformations. Epsilon Transformation Language (ETL) is a hybrid, rule-based Model-to-Model transformation language. ETL provides all the standard features of a transformation language but also provides enhanced flexibility as it can transform many input to many output models, and can query/navigate/modify both source and target models.

Although a number of successful model transformation languages have been currently proposed, the majority of them have been developed in isolation and as a result, they face consistency and integration difficulties with languages that support other model management tasks. ETL, a hybrid model transformation language that has been developed atop the infrastructure provided by the Epsilon model management platform. By building atop Epsilon, ETL is seamlessly integrated with a number of other task specific languages to help to realize composite model management workflows [Errata et al, 2015].

**Kermela**

The Kermeta language was initiated by Franck Fleurey in 2005 within the Triskell team of IRISA. The Kermeta language borrows concepts from languages such as MOF, OCL and QVT, but also from BasicMTL, a model transformation language implemented in 2004 in the Triskell team by D. Vojtisek and F. Fondement. It is also inspired by the previous experience on MTL, the first transformation language created by Triskell, and by the Xion action language for UML. Kermeta, and its execution platform are available under Eclipse. It is open-source, under the EPL.

Kermeta is a general purpose modelling and programming language for metamodel engineering which is also able to perform model transformations. It fills the gap of MOF which defines only the structure of meta-models, by adding a way to specify static semantic (similar to OCL) and dynamic semantic (using operational semantic in the operation of the metamodel). Kermeta uses the object-oriented paradigm like Java or Eiffel.

Kermeta is a modeling and aspect oriented programming language. Its underlying metamodel conforms to the EMOF standard. It is designed to write programs which are also models, to write transformations of models (programs that transform a model into another), to write constraints on these models, and to execute them). The goal of this model approach is to bring an additional level of abstraction on top of the "object" level and thus to see a given system like a set of concepts (and instances of concepts) that form an explicitly coherent whole, which one will call a model [Errata et al, 2015].

*Querry/View/Transform language*

The OMG has defined a standard for expressing M2M transformations, called MOF/QVT or in short QVT [Eclipse, 2016]. Eclipse has two extensions for QVT called QVTd (Declarative) and QVTo (Operational/Procedural). QVT Operational component is a partial implementation of the Operational Mappings Language defined by the OMG standard specification (MOF) 2.0 Query/View/Transformation. In long term, it aims to provide a complete implementation of the operational part of the standard . A high level overview of the QVT Operational language is available as a presentation from EclipseCon 2008, Model Transformation with Operational QVT [Macedo and Cunha, 2013].

AToM3 is a Python based tool for multi-paradigm modeling which stands for "A Tool for Multi-formalism and Meta-Modelling"'. The two main tasks of AToM3 are meta-modeling and model-transforming. Meta-modelling refers to the description, or modelling of different kinds of formalisms used to model systems (although we have focused on formalisms for simulation of dynamical systems, AToM3's capabilities are not restricted to these.) Model-transforming refers to the (automatic) process of converting, translating or modifying a model in a given formalism, into another model that might or might not be in the same formalism.

In AToM3, formalisms and models are described as graphs. From a meta-specification of a formalism, AToM3 generates a tool to visually manipulate (create and edit) models described in the specified formalism. Model transformations are performed by graph rewriting. The transformations themselves can thus be declaratively expressed as graph-grammar models.

Some of the meta-models currently available are: Entity-Relationship, Deterministic Finite State Automata, Non-Deterministic Finite State Automata, Petri Nets, Data Flow Diagrams and Structure Charts. Typical model transformations include model simplification (e.g., state reduction in Finite State Automata), code generation, generation of executable simulators based on the operational semantics of formalisms, as well as behavior-preserving transformations between models in different formalisms. Atom3 is supported by a web based tool, but it has no standalone framework or any integration with a framework such as Eclipse [Lara and Vangheluwe, 2002].

Also, there are other model transformation languages and tools which are mostly under-research and academic studies. Some of them are listed below:

HOTs: Just as any other model can be created, modified, augmented by a transformation, a transformation model can itself be instantiated, modified and so on, by a so-called HOT. This uniformity has several benefits: especially it allows reusing tools and methods, and it creates a framework that can be applied recursively (since transformations of transformations can be transformed themselves) [Massito et al, 2009].

GReAT: It is a transformation language in the generic modeling environment (GME). GReAT language is a graphical language for the specification of graph transformations between Domain-Specific Modelling Languages (DSMLs). It consists of three sub-languages:

- the pattern specification language;
- the transformation rule language;
- the sequencing or control flow language.

Additionally, the input and the output languages of a transformation are defined in terms of meta-models. GReAT is not a standalone tool; rather, it is used in conjunction with the Generic Modelling Environment. However, once a transformation has been developed, a standalone executable can be executed outside of GME [Balasubramanian et al, 2007].

Very detailed review of model transformation approaches and technologies is represented in paper. Authors consider all Model-to-Model transformation mentioned about and a number of other languages, namely Henshin,

MOLA, SiTra, Stratego/XT,Tefkat,Tom,UML-RSDS, and VIATRA2 [Errata et al, 2015].

## Related Papers

Papers, related to software model transformation, can be divided to two classes, namely those which make strong contribution in transformation techniques and those that develop analytical tools for designing new and improving existing transformational approaches and techniques.

Detail review of papers, devoting to designing transformation methods and techniques grounded on practical tools and environments is represented in paper [Chebanyuk and Markov, 2016a]. The result of this review is summarizing achievements of researches according MDE promising. List of MDE promising is also represented in paper [Chebanyuk and Markov, 2016a]. Analyzing this review, the requirements to analytical automated method for Model-to-Model transformations, that cover all MDE promising, were formulated.

Also represent review of papers, making strong contribution in development of transformational techniques.

Paper [Greiner et al, 2016] represents a case study dealing with incremental round-trip engineering of UML class models and Java source code.

Described approach tries to prevent information loss during round-trip engineering by using a so called trace model which is used to synchronize the Platform Independent and the Platform Specific Models. Furthermore, the source code is updated using a fine grained bidirectional incremental merge. Also, information loss is prevented by using Javadoc tags as annotations. Case model and code are changed simultaneously and the changes are contradicting, one transformation direction has to be chosen, which causes that some changes might get lost [Greiner et al, 2016].

The contribution of the survey [Seifermann and Groenda, 2016] is the identification and classification of textual UML modeling notations. During the survey, authors found a total of 31 textual UML notations. The classification is aimed to include the user's point of view and support the notation selection in teams. In total, authors found 14 new notations compared to previous surveys: Alf, Alloy, AUML, Clafer, Dcharts, HUTN, IOM/T, Nomnoml, pgf-umlcd, pgf-umlsd, tUML, txtUML, UML/P, and uml-sequence-diagram-dsl-txl. Authors presented each of the twenty categories in detail including objectively checkable conditions that cover the level of UML support, the editing experience, and the applicability in an engineering team.

Paper [Wu, 2016] addresses the issue of generating metamodel instances satisfying coverage criteria.

More specifically, this paper makes the following contributions:

- — A technique that enables metamodel instances to be generated so that they satisfy partition-based coverage criteria.
- — A technique for generating metamodel instances which satisfy graph properties.

A metamodel is a structural diagram and can be depicted using the UML class diagram notation. Thus, the coverage criteria defined for UML class diagram can also be borrowed for metamodels. To facilitate the transformation from class diagrams with OCL constraints to Satisfiability Modulo Theories (SMT) formulas authors use a bounded typed graph as an intermediate representation [Wu, 2016].

Paper [Natschlager et al, 2016] presents concept for Adaptive Variant Modeling (AdaVM). AdaVM is a part of the AdaBPM framework for advanced adaptability and exception handling in formal business process models. In addition, AdaVM considers linking of elements, propagation of changes, and visual highlighting of differences in process variants. Authors showed that graph transformation techniques are well-suited for process variant management and that variants can be automatically created by a few graph transformation rules specifying concrete variations. Authors show that the adaptable approach is less complex regarding the type graph, source graphs, and the number of rules and application conditions. New ideas, expressed in proposed approaches, are (i) the support of variability by restriction and by extension with graph transformation techniques, (ii) linking and propagation of changes, (iii) individual blocking of elements/attributes, and (iv) visual highlighting of differences in process variants.

A review of mathematical foundations for providing realization of model transformation techniques is outlined in [Rabbi et al, 2016].

A review of metamodeling tools is represented in paper [Favre and Duarte, 2016] and several metamodeling frameworks are described.

## Task and Challenges

To propose an approach and software tool for Model-to-Model transformation.

Challenges: the proposed approach should:

- be based on graph transformations to support easy modifying of transformation rules;
- easily allow modifying transformation rules;
- represent of a transformation rules from behavioral software models to class diagrams at metalevel;

Designed software tool should automate the process of Model-to-Model transformation in IBM Rational Software Architect 7.5.5.2;

## Proposed Approach

Step 1: Define transformation rules. These transformation rules serve as instructions to a transformation engine on how to transform a model. They reflect the mapping patterns between source and target models elements. A developer has to identify which elements of a source model correspond to which elements of a target model.

Step 2:Set up an execution environment. Eclipse plugin with QVT-R script is integrated to IBM Rational Software Architect (IBM RSA). IBM RSA version, used in this paper is 7.5.5.2 and is based on Eclipse 3.4 Ganymede. The tool to be used for Model-to-Model transformation is medini QVT. Medini QVT is a tool set for Model-to-Model transformations.

Step 3: Create a transformation script in medini QVT. The transformation rules are defined using QVT-R language in medini QVT environment. QVT-R script consists of transformations, but it is customary to have a single transformation defined per one script. A transformation, in turn, considers a set of relations between elements of source and target software models. These relations define the patterns that are used to match model elements. Each relation touches of source and target domains. They specify model elements that are to be mapped into each other by binding their properties to variables that can be used in OCL expressions. Additionally, relations can have where and when clauses that specify conditions under which a relation holds.

Step 4: Design source model in RSA environment. RSA is a great tool that provides modeling capabilities for a developer to create models. In a project considered in the previous step, create a model. The best way to define elements of this model is to build a diagram. Create a necessary diagram and open a designer. Whenever you drag the notation elements onto the drawing surface the corresponding model elements are generated in a model container.

Step 5: Execute the transformation. Now you can click on the command the launches a designed plugin. A "Transformation Definition" dialog appears. In a "QVT-R script" field type a path to a QVT-R script or use the "Browse…" button to choose it using File Explorer. In an "Input model" field type a path to a file containing the source model created in a previous step, or use the "Browse…" button to choose it using File Explorer. After that click "OK" button to launch a transformation. As a result, a target model file is created in the same folder as the source model file.

## Description of Transformation Rules

A model transformation, in model-driven engineering, is an automated way of modifying and creating models. An example use of model transformation is ensuring that a family of models is consistent, in a precise sense which the software engineer can define. The aim of using a model transformation is to save effort and reduce errors by automating the building and modification of models where possible.

Model transformations can be thought of as programs that take models as input. There is a wide variety of kinds of model transformation and uses of them, which differ in their inputs and outputs and also in the way they are expressed [Kuster, 2011].

A model transformation usually specifies which models are acceptable as input, and if appropriate what models it may produce as output, by specifying the metamodel to which a model must conform [Rebull et al, 2007].

Consider UML diagram or software model as a graph, where UML diagram objects are graph vertices and UML diagram links are edges of this graph. Thus, consider an analytical background for representing transformation rules. Denote software model(SM) as: SM(O,L), where

O – a set of software model objects. Objects are elements of software model (SM) notations that can be expressed as graph vertexes.

L – a set of links between O, that can be expressed as graph edges. Links are elements of software model notation that can be expressed as edges [Wiemann, 1995].

As in transformation operation there are two software models define them as initial ($SM_{initial}$) and resulting ($SM_{resulting}$). $SM_{initial}$ is a software model from which transformation is started. This model contains initial information for transformation. $SM_{resulting}$ is the model which is obtained after transformation. Thus:

$$\begin{aligned}
&SM_{initial} = (O_1, L_1); O_1 = \{o_{1,i} \mid i = 1,...,n_1\}; \\
&L_1 = \{l_{1,j} \mid j = 1,...,m_1\};\ n_1 = |O_1|;\ m_1 = |L_1| \\
&SM_{resulting} = (O_2, L_2); O_1 = \{o_{2,k} \mid k = 1,...,n_2\} \\
&L_2 = \{l_{2,p} \mid p = 1,...,m_2\};\ n_2 = |O_2|;\ m_2 = |L_2|
\end{aligned} \qquad (1)$$

where $O_1$ - set of $SM_{initial}$ objects, $O_2$ - set of $SM_{resulting}$ objects.

$L_1$ - set of $SM_{initial}$ links. $L_2$ - set of $SM_{resulting}$ links.

*Initial* and *resulting* are types of software models. For example if transformation performed from use case to communication diagram we write transform $SM_{use\_case}$ to $SM_{collaboration}$.

To represent transformation rules transformational grammar [Chomsky, 1957] is used. Transformation rules are syntax of this grammar [Chebanyuk and Markov, 2016a].

Denote transformation operation as " $\rightarrow$ " [Chomsky, 1957]. Thus, transformation operation from $SMI_{sub}$ to $SMR_{sub}$ is written as follows:

$$SM_{initial} \rightarrow SM_{resulting} \qquad (2)$$

Expression (5) means that applying set of transformation rules, sub-graphs, formed from initial software models are transformed to sub-graphs of resulting software model.

Considering graphs as a set of objects and links, the next expression can be written:

$$\begin{aligned}
&(OI, LI) \rightarrow (OR, LR); \\
&OI \subset O_1,\ OR \subset O_2,\ LI \subset L_1,\ LR \subset L_2
\end{aligned} \qquad (3)$$

Represent denotation for Behavioral Software Model (BSM). they were introduced in paper [Chebanyuk, 2015]. Table 1 illustrates denotation for different BSM object.

XMI representation, based on abstract syntax tree concept, allows representation all UML diagrams in unified format.

Using this fact let us formulate one-to-one transformation rule for all elements of different UML diagram notation, that mean the same role of business processes. ***UML diagrams objects that mean the same and have different names in different UML diagrams are transformed from source UML diagram to target one by means of mapping one-to-one.***

Table 2 shows that when objects play the same role in different BSM they are transformed one-to-one. They could be named the same (for example $Y_C$ is transformed to $Y_S$) or having different names (for example $P_{UC}$ is transformed to $M_S$).

**Table 1**. Denotations of BSM objects

| Denotation from the UML standard | Denotation of the proposed approach | Use case | Collaboration | Sequence |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| Object | O | - | + | + |
| 1 | 2 | 3 | 4 | 5 |
| Complex object (object with properties defined) | $O_{COM}$ | - | + | - |
| Actor | A | + | + | + |
| Message | M | + (precedent is an equivalent of message) | + | + |
| Collection | Y | - | + | + |
| Subsystem | SS | + | + | + |
| Multiplicity | N | + | + | - |
| Waiting for response | I | - | - | + |

**Table 2**. One-to-one Model-to-Model transformation mappings

| Denotation from the UML standard | Denotation of the proposed approach | Use case (UC) | Collaboration © | Sequence (S) |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| Object | O | - | $O_C \rightarrow O_S$ | $O_S \rightarrow O_C$ |
| Complex object (object with properties defined) | $O_{COM}$ | - | $O_{COM} \rightarrow O_s$ | - |
| Actor | A | $A_{UC} \rightarrow A_C$ $A_{UC} \rightarrow A_S$ | $A_C \rightarrow A_{UC}$ $A_C \rightarrow A_S$ | $A_S \rightarrow A_{UC}$ $A_S \rightarrow A_C$ |
| Message (precedent) | M (P) | $P_{UC} \rightarrow M_C$ $P_{UC} \rightarrow M_S$ | $M_C \rightarrow P_{UC}$ $M_C \rightarrow M_S$ | $M_S \rightarrow P_{UC}$ $M_S \rightarrow M_C$ |
| Collection | Y | - | $Y_C \rightarrow Y_S$ | $Y_S \rightarrow Y_C$ |
| Subsystem | SS | $SS_{UC} \rightarrow SS_C$ $SS_{UC} \rightarrow SS_S$ | $SS_C \rightarrow SS_{UC}$ $SS_C \rightarrow SS_S$ | $SS_S \rightarrow SS_{UC}$ $SS_S \rightarrow SS_C$ |

## Setting up execution environment

The primary tool to be used for Eclipse plugin development that focuses on modeling software is IBM RSA. IBM Rational Software Architect is a modeling and development environment that uses the UML for designing architecture for C++ and JEE applications and web services. Rational Software Architect is built in the Eclipse open-source software framework and includes capabilities focused on architectural code analysis, C++, and MDD with the UML for creating applications and web services [IBM, 2015]. Used version is Rational Software Architect 7.5.5.2 that is based on Eclipse 3.4 Ganymede. Since RSA is Eclipse-based, it can take advantage of the market of third-party plugins for Eclipse, as well as plugins specifically for Rational tools.

The tool to be used for Model-to-Model transformation is medini QVT [OMG, 2015], [QVT, 2016]. Medini QVT is a tool set for Model-to-Model transformations [Medini QVT, 2016]. The core engine implements OMG's QVT Relations standard, and is licensed under EPL. Medini QVT also includes tools for convenient development of transformations, such as a graphical debugger and an editor. These components are freely available for non-commercial use only.

Medini QVT has the following features:

– execution of QVT transformations expressed in the textual concrete syntax of the Relations language;
– Eclipse integration;
– editor with code assistant;
– debugger to step through the relations;
– trace management enabling incremental update during re-transformation.

Key concept enabling incremental update as well as the transition from manual modeling to automation through transformations in the absence of traces.

It is a challenge to integrate medini QVT into RSA since simply copying the plugins into the “plugins” folder will not make RSA discover them. The less efficient and safe solution is to use a separate Eclipse 3.6 Helios distribution and install medini QVT in there from an update site. Then in RSA the folder with the Eclipse 3.6 distribution can serve as a local repository of plugins for RSA. By navigating to “Help” menu, and choosing “Software Updates…” option we can switch to “Available Software” tab and click “Add Site…” button. There we can point to an Eclipse distribution with medini QVT installed by clicking “Local…” button. Then in an appeared site option we can chose “QVT Cockpit Feature” and click “Install…”. After that it is a simple matter of clicking “Next” or “Finish”.

The project is an Eclipse plugin project that has several dependencies. Among them are the following required plugins:

**Table 3 Description of used Eclipse plugins to plug-in project dependencies**

| Plugin name | Plugin features |
|---|---|
| 1 | 2 |
| org.eclipse.ui | Application programming interfaces for interaction with and extension of the Eclipse Platform User Interface. |
| org.eclipse.core.runtime | Provides support for the runtime platform, core utility methods and the extension registry. |
| com.ibm.xtools.modeler.ui | UML Modeler primary package. This package exposes the entry point for the UML Modeler API though UMLModeler static class. |
| de.ikv.medini.qvt.plugin | Contains QVT-R transformation engine. |

**Table 4. Description of imported packages plug-in project dependencies**

| Package name | Package features |
|---|---|
| com.ibm.xtools.modeler.ui | The UML Modeler API consists of a single static utility class, UMLModeler, and of several other classes and interfaces that are accessible from UMLModeler. |
| com.ibm.xtools.uml.ui.diagram | This package provides the UML modeling API that enables UML diagrams to be created and modified. |
| com.ibm.xtools.umlnotation | UML Notation meta-model primary package. |
| org.eclipse.emf.common.util | Provides basic utilities. |
| org.eclipse.emf.ecore | Provides an API for the Ecore dialect of UML. |
| org.eclipse.emf.ecore.resource | Provides an API for modeling abstract persistent resources. |
| org.eclipse.emf.ecore.resource.impl | Provides an implementation of the resource API. |
| org.eclipse.emf.ecore.xmi.impl | Provides implementation of XMI utility |
| org.eclipse.gmf.runtime.notation | Provides a standard EMF notational meta model. |
| org.eclipse.uml2.uml | Provides an API for an EMF-based implementation of the UML 2.5 metamodel for the Eclipse platform |

## Desciption of Medini QVT transformation script

Let’s consider a use case model first. Just as a class model use case model has a root level contained called Package. Naturally it can be mapped into a Package within a class model. Then we might consider an Actor. Actor can be mapped into a Class that implements an Interface. This implementation reflects a Realization relationship element. Then there is a UseCase. Being connected to an Actor through the Association relationship it can represent an Operator owned by an Interface.

The following table summarizes the rules of transformation from use case model to class model:

**Table 5. Transformation rules one-to-one from UML Use Case Diagram to class one**

| Use Case Model | Class Model |
|---|---|
| Package | Package |
| Actor | Class, Interface, Realization |
| UseCase | Operation |

The second case of a source model is communication model. Yet again we consider a Package element that is mapped into a similar Package element in a target class model. Then a Lifeline of types Actor, Class, and Component can be mapped into a Class of a class model. Then a Message passed between lifelines can be mapped to an Operation inside of a Class.

The following table summarizes the rules of transformation from communication model to class model:

**Table 5. Transformation rules one-to-one from UML Communication Diagram to class one**

| Communication Model | Class Model |
|---|---|
| Package | Package |
| Lifeline (Actor, Class, Component) | Class |
| Message | Operation |

## Description of QVT-R Script

Here is a short overview of the integral classes written for a plugin application to support its functionality.

A custom class that represents a transformation engine is called Medini QVTTransformationEngine and it implements custom TransformationEngine interface. It relies on third-party EMFQvtProcessorImpl class to evaluate QVT transformation. An abstract custom class ModelTransformationTemplate conforms to a Template design pattern. It encapsulates a skeleton of an algorithm that describes the model transformation process. The algorithm consists of four steps:

– prepare source model placeholder;
– prepare target model placeholder;
– perform transformation;
– obtain target model.

These four steps are represented by methods and these methods are invoked in a template method named as transformSourceModel of a ModelTransformationTemplate class.

A custom concrete class that extends ModelTransformationTemplate is called Medini QVTRsaDomModelTransformation. The "Medini QVT" part of a class name means that the transformation process is performed by this model transformation tool. The "Rsa" part of the name means that the source model is provided by the Rational Software Architect and a diagram in a target model is created using this tool also. The "Dom" part in the name specifies that the DOM method of XML editing is used.

```
transformation usecase2class (source : uml, target : uml) {
        top relation model2model {
                theName : String;
                checkonly domain source s: uml::Package {
                        name = theName
                };
                enforce domain target t: uml::Package {
                        name = 'ClassModel_from_' + theName
                };
                where {
                        actor2class(s, t);
                }
        }
        relation actor2class {
                modelName : String;
                checkonly domain source sourcePackage: uml::Package {
                        packagedElement = actor: uml::Actor {
                                name = modelName
                        }
                };
                enforce domain target targetPackage: uml::Package {
                        packagedElement = class: uml::Class {
                                name = modelName + 'Impl'
                        },
                        packagedElement = interface: uml::Interface {
                                name = modelName
                        },
                        packagedElement = realization: uml::Realization {
                                client = client -> append(class),
                                supplier = supplier -> append(interface)
                        }
                };
                where {
                        usecase2operation(actor, sourcePackage, interface);
                }
        }
        relation usecase2operation {
                usecaseName : String;
                checkonly domain source actor : uml::Actor {
                };
                checkonly domain source s : uml::Package {
                        packagedElement = usecase : uml::UseCase {
                                name = usecaseName
                        },
                        packagedElement = association : uml::Association {
                                ownedEnd = usecaseEnd : uml::Property {
                                        name = usecaseName.firstToLower()
                                },
                                ownedEnd = actorEnd : uml::Property {
                                        name = actor.name.firstToLower()
                                }
                        }
                };
                enforce domain target interface : uml::Interface {
                        ownedOperation = operation : uml::Operation {
                                name = usecaseName
                        }
                };
        }
}
```

The following listing describes how to transform a communication model into class model:

```
transformation communication2class (source : uml, target : uml) {
        top relation package2package {
                theName : String;
                checkonly domain source s : uml::Package {
                        name = theName,
                        packagedElement = collaboration : uml::Collaboration {
                        }
                };
                enforce domain target t : uml::Package {
                        name = 'ClassModel_from_' + theName
                };
                where {
                        lifeline2class(collaboration, s, t);
                }
        }
        relation lifeline2class {
                modelName : String;
                checkonly domain source collaboration : uml::Collaboration {
                };
                checkonly domain source sourcePackage : uml::Package {
                        packagedElement = lifeline : uml::Classifier {
                                name = modelName
                        }
                };
                enforce domain target targetPackage : uml::Package {
                        packagedElement = class : uml::Class {
                                name = modelName
                        }
                };
               when {
                        lifeline.oclIsKindOf(uml::Actor) or lifeline.oclIsKindOf(uml::Class);
                }
                where {
                        message2operation(lifeline, collaboration, class);
                }
        }
        relation message2operation {
                operationName : String;
                checkonly domain source lifeline : uml::Classifier {
                };
                checkonly domain source collaboration : uml::Collaboration {
                        ownedBehavior = interaction : uml::Interaction {
                                ownedConnector = link : uml::Connector {
                                        end = endPoint : uml::ConnectorEnd {
                                                role = attribute
                                        }
                                },
                                message = theMessage : uml::Message {
                                        name = operationName,
                                        connector = link
                                }
                        },
                        ownedAttribute = attribute : uml::Property {
                                type = lifeline
```

```
                                }
                        };
                        enforce domain target class : uml::Class {
                                ownedOperation = operation : uml::Operation {
                                        name = operationName
                                }
                        };
                        when {
                                link.allOwnedElements()->at(2) = endPoint;
                                not class.allOwnedElements() ->
                                select(x | x.oclIsKindOf(uml::Operation)) ->
                                collect(y | y.oclAsType(uml::Operation)) ->
                                exists(o | o.name = operationName);
                        }
                }
}
```

**Execution example**

Explain how to perform the next points of proposed approach by means of designed script.

Figure1 reflects a dialog window that is part of the plugin and serves as a graphical user interface for the interaction with a user. Such dialog can be created using a standard Eclipse plugin development project.

It is better to execute transformations described by a QVT-R script in an environment that is used for the creation of models. A modeling tool for this purpose is IBM Rational Software Architect. The version used in this paper is IBM RSASE 7.5.5.2. Since Rational Software Architect does not support UML-to-UML transformations there is a need to develop a plugin for the automation of such a task. The plugin has a dependency on another plugin that contains medini QVT transformation engine. So there is a need to integrate the engine into Rational Software Architect.

Since medini QVT 1.7 is based on Eclipse 3.6 Helios and IBM Rational Software Architect 7.5.5.2 is based on Eclipse 3.4 Ganymede the two of them are incompatible. The solution is to install medini QVT into a separate Eclipse 3.6 distribution. Then using this distribution create a feature that would include a plugin containing transformation engine. After that create a local update site and add created feature into it. This update site can be used by Rational Software Architect to install created feature and therefore integrate desired transformation engine for the use by other plugin.

The plugin that transforms UML models with a help of medini QVT transformation engine requires special project structure in order to transform models. Let’s a consider a project folder in RSA environment where the input and output of a transformation are going to be stored. This folder should have three subfolders called “qvt”, “temp” and “traces”. The “qvt” folder is supposed to contain necessary QVT-R scripts. The “temp” folder is supposed to contain temporary file placeholders for source and target models. The “traces” folder will contain the files that are called traces. Traces are generated after the transformation and they store the records of how the transformation took place and which elements participated in a transformation for the creation of target model elements.

The first input field requires a path to a script that describes the rules for the transformation. This must be a QVT-R script. The second field requires a path to an input model to be transformed. The "Browse…" buttons help to find necessary file by opening a file dialog. The "OK" button starts the transformation. "Cancel" button is self-explanatory.

In order to provide an input model into the process of transformation a communication diagram is built based on the plugin's architecture.

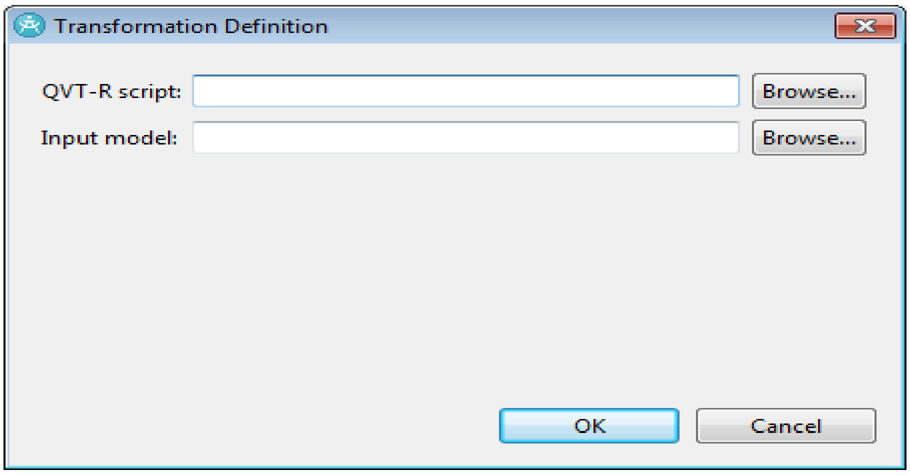

Figure 1. Plug-in window

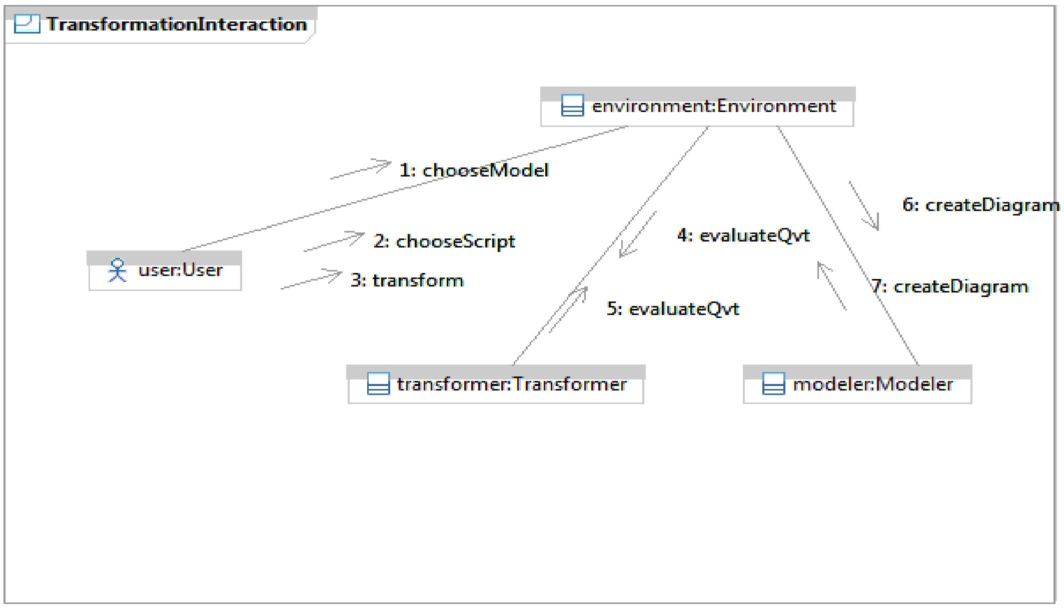

Figure 2. Communication diagram

Communication diagram, represented in Figure 2 is a source model. Particularly there is a “User” actor and three classes “Environment”, “Transformer” and “Modeler”. Each of them have their own type. The interaction starts with the User lifeline being connected to the Environment lifeline, and in turn Environment is connected to the Transformer and Modeler lifelines. Connections are represented by the message paths through which the messages can be passed back and forth. Specifically, User can send such messages as “chooseModel”, “chooseScript” and “transform”. Environment can send “evaluateQvt” message to Transformer and “createDiagram” message to Modeler.

The obtained target model represents a class model with generated class diagram.

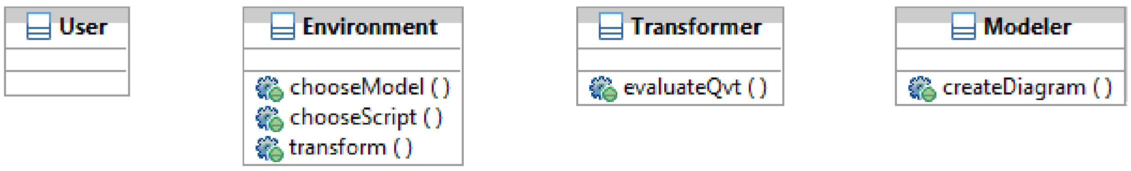

Fig. 3. Class diagram

In general, there are apparent relationships between lifelines being mapped into classes and messages being mapped into operations.

## Conclusion

An approach for Model-to-Model transformation and software tool for class diagram designing are represented in this paper. All variants for one-to-one transformation for UML Use Case, Communication and Sequence diagrams are considered in tables 1 and 2. Such transformations provide a foundation for implementing QVT-R scripts in Medini QVT. The template for QVT-R script for performing one-to-one transformation is proposed in this paper. Using these foundations and QVT-R script templates for one-to-one transformation other transformations can be implemented. An approach for implementation of Model-to-Model transformation when medini QVT plug-ins are built in the IBM Rational Software Architect 7.5.5.2 is represented. Source and target software models are opened in IBM RSA 7.5.5.2.

Designed software tool allows automating the task of creating a class diagrams from the communication diagrams. Flexibility of designed tool allows changing rules of script and scripts themselves (fig. 1) for interoperating with software models of different types. It allows to high skill specialist to formulate specific transformation rules that reflect peculiarities of problem domain and experience of a specialist.

Other advantage of designed plug-in that source and target software models are defined as external parameters for plug-in execution.

Implementing this plugin-in into software development life cycle increases the effectively of all software development where Model-to-Model transformation is performed.

## Further Research

Design an approach and software tool, supporting Model-to-Model transformation, integrating all types of transformation rules, namely many-to-many, one-to-many, and many-to-one. It allows to realize ideas, presented in the papers [Chebanyuk, 2014a] and [Chebanyuk, 2014b]. Integrate transformation from Communication diagram to class one and verification of obtained class diagram by means of analytics, represented in [Chebanyuk and Markov, 2016b].

## Acknowledgements

The paper is published with partial support by the ITHEA ISS (www.ithea.org) and the ADUIS (www.aduis.com.ua)

## Bibliography

[Balasubramanian et al, 2007] Balasubramanian, D., Narayanan, A., van Buskirk, C., Karsai, G. The graph rewriting and transformation language: GReAT. *Electronic Communications of the EASST*, *1, 2007*.

[Chebanyuk and Markov, 2016a] Elena Chebanyuk, Krassimir Markov. Model of problem domain "Model-driven architecture formal methods and approaches". *International Journal "Information Content and Processing", Vol. 2, No.3, ITHEA 2016, p.202-222.*

[Chebanyuk and Markov, 2016b] Elena Chebanyuk, Krassimir Markov (2016). An Approach to Class Diagrams Verification According to SOLID Design Principles.In Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD, ISBN 978-989-758-168-7, pages 435-441. DOI: 10.5220/0005830104350441

[Chebanyuk, 2014a] Chebanyuk E. Method of behavioral software models synchronization. *International Journal "Informational models and analysis" – 2014, №2, pp. 147-163. http://www.foibg.com/ijima/vol03/ijima03-02-p05.pdf*

[Chebanyuk, 2014b] Chebanyuk Elena. An approach to class diagram design. 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Lisbon, Portugal, 2014, pp. 448-453. Access mode:

[Chebanyuk, 2015] Chebanyuk Elena. An approach to behavioral software models analytical representation. *International journal Informational models and analysis, 2015, Vol.3, №1, pp. 61-79 http://www.foibg.com/ijima/vol03/ijima03-02-p05.pdf*

[Chomsky, 1957] Chomsky, Noam. Syntactic Structures. Mouton publishers, Eilenberg: Mac Lane The, Hague*, 1945 - 1957. ISBN 90 279 3385 5. 1957 p.107. http://ewan.website/egg-course-1/readings/syntactic_structures.pdf*

[Cicchetti et al, 2010] Cicchetti, A., Di Ruscio, D., Eramo, R., & Pierantonio, A. (2010, October). JTL: a bidirectional and change propagating transformation language. In International Conference on Software Language Engineering (pp. 183-202). Springer Berlin Heidelberg.

[Eclipse, 2016] ATL Transformation Language http://www.eclipse.org/atl/

[Errata et al, 2015] Ferhat Errata, Moharram Challenger, Geylani Kardas, Moharram Challenger, Geylani Kardas, Review of Model-to-Model Transformation Approaches and Technologies, 2015 Mode of access: https://itea3.org/project/workpackage/document/download/2305/13028-ModelWriter-WP-3-D311ReviewofModel-to-ModelTransformationApproachesandTechnologies

[Favre and Duarte, 2016] Favre, Liliana and Daniel Duarte. 2016. Formal MOF Metamodeling and Tool Support. *In: MODELSWARD 2016, Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development. Edited by S. Hammoudi, L.F. Pires, B. Selic and P. Desfray. SCITEPRESS – Science and Technology Publications, Lda. Portugal, 2016. ISBN: 978-989-758-168-7. pp. 99-110. DOI:10.5220/0005689200990110, http://www.scitepress.org/DigitalLibrary/ProceedingsDetails.aspx?ID=j1i7qrX33Ns=&t=1*

[Greiner et al, 2016] Greiner Sandra, Thomas Buchmann, Bernhard Westfechtel. 2016. Bidirectional Transformations with QVT-R: A Case Study in Round-trip Engineering UML Class Models and Java Source Code. *In: MODELSWARD 2016, Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development. Edited by S. Hammoudi, L.F. Pires, B. Selic and P. Desfray. SCITEPRESS – Science and Technology Publications, Lda. Portugal, 2016. ISBN: 978-989-758-168-7. pp. 15-27. DOI:10.5220/0005644700150027 http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=efZXth7Zbbg=&t=1*

[IBM, 2015] IBM Rational Software Architect (2015) – Mode of access: https://en.wikipedia.org/wiki/Rational_Software_Architect

[Kuster, 2011] Jochen Küster. Model-Driven Software Engineering Model Transformations II IBM Research (2011) – Zurich. access mode http://researcher.ibm.com/researcher/files/zurich-jku/mdse-07.pdf

[Lara and Vangheluwe, 2002] Lara J., Vangheluwe H. (2002) AToM3: A Tool for Multi-formalism and Meta-modelling. In: Kutsche RD., Weber H. (eds) Fundamental Approaches to Software Engineering. FASE 2002. Lecture Notes in Computer Science, vol 2306. Springer, Berlin, Heidelberg

[Macedo and Cunha, 2013] Macedo, N., Cunha, A. Implementing QVT-R bidirectional model transformations using Alloy. In *International Conference on Fundamental Approaches to Software Engineering* 2013 (pp. 297-311). Springer Berlin Heidelberg.

[Massito et al, 2009] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., & Bézivin, J. (2009, June). On the use of higher-order model transformations. In European Conference on Model Driven Architecture-Foundations and Applications (pp. 18-33). Springer Berlin Heidelberg. 2009

[Medini QVT, 2016] access mode http://projects.ikv.de/qvt

[Natschlager et al, 2016] Natschlager, Christine, Verena Geist, Christa Illibauer and Robert Hutter. 2016. Modelling Business Process Variants using Graph Transformation Rules *In: MODELSWARD 2016, Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development. Edited by S. Hammoudi, L.F. Pires, B. Selic and P. Desfray. SCITEPRESS – Science and Technology Publications, Lda. Portugal, 2016. ISBN: 978-989-758-168-7. pp. 65-74. DOI:10.5220/0005686900870098, http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=IzjjeczBZuA=&t=1*

[OMG, 2015] XML Metadata Interchange http://www.omg.org/spec/XMI/2.5.1/

[QVT, 2016] Meta Object Facility Query/View/Transformation, v1.3

[Rabbi et al, 2016] Rabbi, Fazle, Yngve Lamo, Ingrid Chieh Yu, Lars Michael Kristensen. 2016. WebDPF: A Web-based Metamodelling and Model Transformation Environment. *In: MODELSWARD 2016, Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development. Edited by S. Hammoudi, L.F. Pires, B. Selic and P. Desfray. SCITEPRESS – Science and Technology Publications, Lda. Portugal, 2016. ISBN: 978-989-758-168-7. pp. 87-98. DOI:10.5220/0005686900870098, http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=IzjjeczBZuA=&t=1*

[Rebull et al, 2007] Rebull, E. V. S., Avila-Garcıa, O., Garcıa, J. L. R., Garcıa, A. E. (2007). Applying a modeldriven approach to model transformation development.

[Seifermann and Groenda, 2016] Seifermann, Stephan and Henning Groenda. Survey on Textual Notations for the Unified Modeling Language *In: MODELSWARD 2016, Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development. Edited by S. Hammoudi, L.F. Pires, B. Selic and P. Desfray. SCITEPRESS – Science and Technology Publications, Lda. Portugal, 2016. ISBN: 978-989-758-168-7. pp. 28-39. DOI:10.5220/0005686900870098,*

[Wiemann, 1995] Wiemann, H. (1995). Theory of Graph Transformations access mode: http://www.informatik.uni-bremen.de/st/lehre/Arte-fakt/Seminar/Ausarbeitungen/04_Theorie_Graphtransformationen.pdf

[Wu, 2016] Wu, Hao Generating Metamodel Instances Satisfying Coverage Criteria via SMT Solving *In: MODELSWARD 2016, Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development. Edited by S. Hammoudi, L.F. Pires, B. Selic and P. Desfray. SCITEPRESS – Science and Technology Publications, Lda. Portugal, 2016. ISBN: 978-989-758-168-7. pp. 40-51. DOI:10.5220/0005686900870098,*

## Authors' Information

***Elena Chebanyuk** – Assoc. Prof. of Software Engineering Department, National Aviation University, Kyiv, Ukraine,*

***e-mail**: chebanyuk.elena@ithea.org*

***Major Fields of Scientific Research**: Model-Driven Architecture, Model-Driven Development, Software architecture, Mobile development, Software development*

***Kyryl Shestakov** – Software Engineering Department, National Aviation University, Kyiv, Ukraine,*

***e-mail**: KyrylO.Shestakov@livenau.net*

***Major Fields of Scientific Research**: Model-Driven Architecture, Model-Driven Development, Software architecture, Mobile development, Software development*