Automated Model Transformation in MDA

© Mikhail Kuznetsov

Computational Mathematics and Cybernetics Faculty,
Moscow M.V.Lomonosov State University
mikle.kuz@mtu-net.ru
Ph.D. Advisor S.D. Kuznetsov

Abstract

Model Driven Architecture is a new and promising approach to software development. But its spread is hindered by the fact that one of its parts - automated transformation of software models – is not sufficiently developed. In this paper a language is presented that can be used to define such transformations and a tool to automatically execute them.

1 Introduction

Programmers use many different middleware platforms and technologies, and in the future their number will only increase because new technologies are being developed constantly but old ones become obsolete very slowly. All attempts to create a universal platform that could replace all existing ones resulted only in increased variety of technologies. The problem of choosing a platform for a particular project becomes more and more important, as well as problems of interaction and integration of heterogeneous systems and migration of existing systems to a new platform when old platform becomes outdated and no longer satisfies customer’s needs. The solution may be usage of a new methodology of software development.

Object Management Group (OMG) offers a new approach to software development called Model Driven Architecture (MDA) [9]. MDA offers several advantages compared to existing methodologies: simplification of development of multi-platform systems, ease of switching of a middleware platform, increased speed of development and quality of products and much more. But all of it is possible only when development tools will support the MDA technology and help to fully realize its potential. Unfortunately currently most of commercial products, even those that claim to support MDA, do not offer proper tools and technologies, thus efficiency of MDA in real projects is limited.

MDA is based on concepts of Platform-Independent and Platform-Specific Models (PIMs and PSMs) [12].

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYRCoDIS, St.-Petersburg, Russia, 2005

First during development a PIM is created. PIM is a model of a system that contains its business-logic, usecases and view of a system from end user’s point of view without details of its actual implementation. The fact that PIM is not bind to any platform or technology is most innovative and most important. When using MDA it is recommended to develop platform-independent model with a relatively high level of details, up to using a high-level platform-independent programming language to code system’s functionality and creating an executable prototype.

Once PIM has sufficient details, a transition to Platform-Specific Model is performed. This model describes not only user-level system functionality, but also details of implementation of the system on the middleware platform chosen for a current project. More details are being added to the model by developers, and necessary changes are performed, until the model is ready to be passed to the stage of code generation. Just as in a common development process, a code can be partially generated automatically from the model, and then finalized manually and compiled.

Of course, actual development process is not so straightforward. It is nearly impossible for a complex project to make a platform-independent model that would not require any modifications on later stages of development. During development of platform-specific model and even of code changes can be made in any higher-level models. It does not contradict MDA development process, but when making such changes one should keep correspondence between models: a change to one model should be properly reflected in all others. So, when using MDA three models of the systems are being developed simultaneously: PIM, PSM and code, each reflecting its own level of abstraction.

The idea that is placed in the core of MDA is independent from modeling language and tools. But the developer of technology, OMG consortium, assumes that modeling will be done with UML (Unified Modeling Language) [6]. Last changes and additions to UML standard made this language much more convenient for use with MDA. UML Action Semantics [13] allows to describe system’s functionality on platform-independent level, and UML Profiles make creation of PSMs for specific middleware platforms easier.

Separation of platform-independent and platform-specific models offers major benefits for developers. First of all, the process of transition between middleware platforms and technologies becomes easier, because PIM can be reused and only PSM has to be created anew. A risk of early development mistakes decreases, because it is much easier to find and correct such mistakes on relatively simple model that is fully based on customer's requirements, then on sophisticated low-level model that contains huge number of implementation-specific details. Separation of the models is also useful for fast creation of documentation, integration, creation of heterogeneous systems and so on.

But the main advantage of MDA is its ability to increase development speed, despite the fact that two models are being made instead of one. This is achieved by using automated generation of platform-specific model by platform-independent one. The process of transition from PIM to PSM that is based on a certain technological platform is highly formalized. UML Profiles that are developed for all popular middleware platforms contain recommendations about mapping various UML elements to forms that are specific to a chosen technology – but those are recommendations for a developer and not instructions for automated execution. With a bit more efforts and more formalization it is possible to convert loose form of Profiles into exact instructions. Then it is no longer needed to manually create PSMs, and development speed will increase significantly, because many details required by a certain technology will be added to the model automatically. Besides, the number of mistakes that are inevitable with manual modeling will decrease as well. Description of automatic transition to a PSM can be made once and thoroughly tested, and then used in all projects that use this technology. So, when using MDA and automatic transition to platform-specific model, the development process consists of the following stages:

- creation of the task, usecases, requirements and other initial documents;
- creation of platform-independent model;
- automatic transition from PIM to platform-specific model, using a standard transformation definition developed earlier (and probably by other company) for a chosen middleware platform and/or implementation technologies;
- manual modification of both PIM and PSM, addition of various details;
- automated code generation;
- manual coding of parts that could not be generated automatically, compilation.

Transition from PSM to code is fairly well developed. Before MDA it was called "code generation" and to a certain degree can be performed by almost any modeling tool for majority of programming languages and technologies. But automated transition to PSM is a new concept. Since PIM and PSM are both models written on UML (at least with classic MDA approach), transition is just a transformation of a UML-model using a pre-defined transformation definition (that contains formal description of details of a specific technology or platform) [8]. This paper is dedicated to development of a language and tool for defining and executing such transformations.

There are several requirements that a transformation language has to satisfy to be efficient when used in MDA:

- Formalism of transformation description. Transformation should be defined in a formal language that has a good grammatical model, so that the transformation tool could interpret this language and automatically execute it.
- Universality. Transformation language should permit creation of a wide range of transformations that can cover a variety of middleware platform, including those that will appear in the future. Then it will always be possible to pick a standard transformation definition and transform a PIM of a project to a modern platform, even if the PIM was made 10 years ago and the platform was developed recently. It is desirable that the language contains means for model parameterisation and tuning, so that a single transformation definition can be used in many different projects.
- Integrity preservation. During development all models can be changed, even after transition from PIM to PSM, meaning that there should be a way to automatically keep conformity between transformed models that has been achieved during transformation. This means that information should be kept about a course of transformation and about mapping that was established between different elements of the models. Then UML editing tool can use this information to automatically map changes done to one model to another one, or at least to notify a programmer what elements are no longer consistent.
- Intelligibility for a human reader. One transformation definition can be used in many projects, but all projects are different. This means that it should be relatively easy to understand and modify it to suit specific needs. The transformation should be clear not only to the person who wrote it, but also to other people who may have to alter it. It is also desirable that the transformation is well-structured, so that it is easier to understand what effect a particular change will have on transformation globally.
- Interconnected transformations. It is possible that during MDA-based development multiple models of one type exist in a project. In particular, if the project uses multiple platforms, a separate PSM is generated for each of them. Transformation language should be able to operate with more then one source and generated model, it should be possible to transform multiple models within a single transformation definition. It is much more convenient than usage of a separate transformation for each pair of models, since such approach allows

to easily track relation between model elements and generate any necessary mediators and bridges.

2 Various approaches to model transformation

There are multiple approaches to definition and execution of transformation of UML models [3]. The simplest solution is to imperatively define the procedure of transformation using any algorithmic language. A UML modeling toolset may contain modules that contain predefined transformations that can later be used when necessary. Unfortunately such an approach is badly suited for MDA-style development. First of all, users cannot add their own transformation definitions or modify those provided by creators of a toolset. Also each toolset will handle transformations differently even for the same middleware platforms, meaning that models created by different toolsets will likely be incompatible with each other. Instead of being restricted by choice of a middleware platform, a programmer will be restricted by choice of a modeling toolset. Finally, such an approach means that for each toolset a full number of transformations should be developed for each middleware technology. This is a huge work compared to using a single set of transformation created and thoroughly tested by a third party.

Another approach to transformation is usage of well-developed concepts and ideas from other areas of science. In particular, it is possible to present a UML model as a graph and use graph transformation technologies. The main disadvantage of this approach is the fact that it uses its own terms and definitions that are not related to modeling. User of such transformation tool will have to know not only UML modeling, but also graph theory. And every time he wants to make a change in transformation definition he will have to make mental transition to graphs, and then back to UML.

Another solution is to use transformations of XML and XMI standard. XMI [10] (XML Metadata Interchange) is a standard that allows to present UML as an XML document, its main purpose is storage of UML models and exchange of models between different tools. There are several reliable technologies for XML transformation, such as XSLT [14] and XQuery [1]. A UML model can be converted to XML using XMI, then this XML document can be transformed and converted back to UML. But such double conversion makes the transformation definition unclear for a human reader. Also since we need to convert transformed XML document back to UML, it has to comply with XMI. On practice this means that nearly 90% of definition of XML transformation is targeted not at actual model transformation, but at ensuring that the result is a valid XMI representation of a model. Of course it is very hard to make and understand such transformations. Even a relatively simple model transformation is defined by a very large and bulky XML transformation.

UML language is considered to be a universal modeling tool, and of course it contains its own means to define transformations (not UML model transformations, but transformations in general). In particular CWM (Common Warehouse Metamodel) standard has such functionality [2]. The idea to use UML language to define UML transformations, just as UML defines it's own syntax, looks promising, but not very practical. Unfortunately UML is just a modeling language, it can be used to show the fact that there is a mapping between certain model elements, but not to define details of such a mapping in general with enough precision to allow automated execution. Another standard from UML family – QVT (Query, View, Transformation) [7] – is meant to fill this niche and looks much more promising for use in MDA. Unfortunately, currently this standard is in early stages of development, and it is not possible to say when it will be ready or what exactly it will contain. The standard should do several big tasks at the same time, and is mostly targeted not on practical use but on development of theoretical concept of metametamodeling as part of MOF (Meta Object Facility, the standard for representation of metamodels) [11]. It is likely that this standard will be inefficient in practice, and will not satisfy requirements of MDA.

One more solution is development of a specialized high-level language for defining model transformations and a tool for execution of such a language that would be efficient for application in MDA. Probably such a tool will be more convenient then an adaptation of a certain generic standard. Below we'll review one of such languages that is developed by the author.

3 Fundamental scheme of the transformation tool

Transformation tool can be a separate program or a part of a larger software development toolkit. During MDA development it is used for partial automation of generation of a platform-specific model [4]. It receives the following input data:

- one or more source models;
- metamodel for each model that takes part in the transformation;
- transformation definition on a special transformation definition language; the definition depends on the metamodels and on number of models that participate in the transformation, but is independent from specific models.

The output data is:

- a set of source models with modifications added during transformation;
- one or more (depending on transformation definition, possibly zero) newly generated models that were created during a transformation; each generated model has to comply to one of the metamodels supplied as input data;
- information about dependencies and mappings between elements of models that was established during transformation; it is needed to preserve conformity of models after the transformation.

For the transformation tool there is no major difference between source and generated models: both can be modified during transformation, the difference is only that generated model starts as empty model. So in the future we'll talk about set of models meaning source and generated models combined.

4 Transformation definition language

Transformation definition consists of one or more modules. Each module has a unique name and consists of a set of transformation rules. Header of the module can also specify the sequence of execution of rules in the module; this option will be explained later. Below we can see a part of formal definition of the language using extended Backus-Naur form (eBNF).

transformation::= <stage>*;
stage::= **stage** <name> [<sequence>]
{ <transformation_rule>* };
sequence::= [**reversed**]
(**linear** | **loop** | **rollback** | **rulebyrule**);

Each rule has a unique (in the scope of the module) name and consists of select section that defines when the rule can be applied, and generation section, that specifies actions to execute when applying the rule.

transformation_rule::= **rule** <name>
{ <select_section> <generate_section> };

Select section contains a sequence of selection operators. Each operator defines a new variable that is called "*selection variable*". The name of this variable should be unique in the scope of the current rule, and domain is a set of elements of a model specified by *navigation expression*. Besides, select section can contain *qualifying conditions* - logical expressions that can contain selection variables declared by operators that stand earlier in the rule.

select_section::= (<select_operator>|<constraint>)*;
select_operator::=**forall** <name>
from <nav_expression>;
constraint::= **where** <condition>;

Navigation expression is a sequence of directions that begins with a name of an existing variable (we'll call it a base variable of the expression), symbol '/' is used as a separator. Direction is a name of association on a metamodel that corresponds to the model that is being transformed (if more then one model participates in the transformation, the metamodel can be determined by a type of the base variable). Cardinality of a direction is multiplicity of the corresponding metamodel association.

nav_expression::=<name>
iteration_pair(/, <nav_direction>);

Computation of a navigation expression consists of a sequential transition from one model element to another using specified directions, starting from the element that the base variable points at. If a cardinality of a direction is greater then one, all corresponding model elements are considered to be the result. This means that the result of execution of a navigation expression is a list of all model elements that can be reached from the element indicated by base variable by the specified set of directions. Navigation expression has the following properties:

- Type: metamodel element that corresponds to elements contained in the result. Since navigation is performed via metamodel, all elements of the result will always have the same type. Type can be determined statically, it depends on the metamodel but not to the actual model.
- Cardinality: maximum number of elements in the result. Cardinality is determined statically.
- Value: a set of elements that were received as a result during computation of the expression. Value of an expression is calculated during its execution at an actual model.

Navigation expression can begin with any local or global variable declared earlier. Local variable is a variable that is declared in one of operators of the same rule. Global variables are names of models that participate in the transformation. Obviosly navigation expression of the first select operator of a rule always begins with a global variable, since no local variables are initialized yet.

Generation section is a sequence of operators that modify models. Create operator allows to add a new model element where a set of elements is possible (where multiplicity of a corresponding metamodel association is greater then one). Navigation expression in this operator should point at the set, and its type determines the type of created element. Delete operator excludes a model element from a set. Modification operator allows changing value of element's attribute. Two more operators allow adding existing element to a set of elements or removing it from a set without deleting it from the model; those operators are necessary if metamodel contains loops or if the same element is accessible via multiple associations.

generate_section::=(
<create_operator> |
<update_operator> |
<delete_operator> |
<include_operator> |
<exclude_operator>)*;
create_operator::=**make** <name>
in <nav_expression>; ;
update_operator::= <nav_expression> =
<expression>; ;
delete_operator::= **delete** <nav_expression>; ;
include_operator::=**include** <name>
in <nav_expression>; ;
exclude_operator::=**exclude** <name>
in <nav_expression>; ;

5 Execution of a transformation

Once a transformation tool receives a source model (or models) and transformation definition, as well as description of used metamodels, it can execute the transformation. It consists of a sequential execution of modules from the definition. Execution of a module is sequential repetition of two steps: finding a rule that can be applied and applying this rule.

Application of a rule is determined by its select section. Each operator of that section declares a new variable and defines a set of its possible values. The rule can be applied to a set of model elements, where each element is in the appropriate value set, and where all conditions of the select section are true. This means that if a select section has three selection operators and no conditions, and the operators define variables v1,v2 and v3 that can take on N1,N2 and N3 different values, then the rule can be applied to N1*N2*N3 different value sets; if the rule has any conditions then it can be applied only to those value sets where all conditions are true. A rule can be applied only to the same value set only once.

Application of a rule to a value set is execution of all operators in the generation section.

The order of application of rules is determined by a special parameter in the module's header. If its value is 'linear' then rules are applied sequentially from first in module's definition to the last; search for the next rule to apply starts from the last applied rule. The module is considered to be finished once the last rule in the module is executed for all possible value sets. If the value of the parameter is 'loop', the transformation works in the same way, but upon reaching the last rule the search for applicable rules continues from the first rule. The module is finished if no more rules can be applied. If the value of <sequence> parameter is 'rollback', then after each application of a rule the search starts from the first rule in the module; the transformation is over if no more rules can be applied. If the value is 'rulebyrule', then search is also performed from the first rule in the module, but once an applicable rule Is found it is applied to all possible value sets – and only then the search starts anew from the top. All values of parameters mentioned above can also be used with a keyword 'reversed', meaning that the order of rules in the module is reversed and all searches are performed from the last element from module definition to the first. If no value of <sequence> parameter is specified, it is considered to be 'rulebyrule' by default, since it allows fastest execution of the transformation. Transformation is over once the last module is finished.

It should be noted that it is not possible to warrant that a transformation will ever be finished for any model. In some transformations infinite loops may appear, especially when using 'create element' operator. Users of the transformation definition language should keep it in mind when defining and executing a transformation.

6 Transformation bond and its usage in transformation definition

Every time a rule is applied a special data structure called 'transformation bond' is created in addition to any actions contained in the generation section. A name of this structure is the same as rule's name, and its attributes have the same names and types as rule's local variables. Values of those attributes are equal to values of corresponding variables calculated during application of the rule.

Transformation bonds are stored during entire process of a transformation, and possibly even after the transformation is finished. They contain information about a course of the transformation, about application of any rule and about relations between model elements established by the transformation. This information can be used to maintain consistency between models in the process of transformation and after it. Besides, transformation bonds may be used by a transformation tool so that it can warrant that any rule is applied only once to any set of values of selection variables.

It is possible to use transformation bonds explicitly in the transformation definition to establish dependencies between rules. Just as navigation expressions are used to select elements from models, they can be used to select bonds created by rules that were applied before the current rule. Such special navigation expressions begin not with a local variable or a name of a model, but with a name of a transformation module, followed by a name of a rule and then a name of a variable in that rule. Since any rule can be applied more then one time during a transformation, creating more then one instance of a corresponding bond, cardinality of such navigation expression is always 'many'. Just as with an ordinary navigation expression, it returns a set of model elements as a result – a set of elements that were values of a specified variable in all applications of a specified rule.

<special_nav_expression>::=<block_name>/
<rule_name>/<variable_name>;

Instead of the name of a module a keyword 'rules' may be used that is equal to the name of a current module.

Explicit use of transformation bonds in transformation definition gives a powerful tool for a programmer. It allows to define very complex rules in a short and easy to understand way.

7 Rule extension and templates

It is possible to define a new rule as an extension of existing rule. To do so one has to write a name of the rule that is being extended after the name of extending rule in that rule's header. It is possible to extend extensions and to make several extensions of the same

base rule, creating multi-layer hierarchy. Operators in the extending rule can use all variables from base rule along with their own variables (select operators can use variables only from select section and not from generation section).

transformation_rule::= **rule** <name> [**:** <name>] {
<select_section> **;** <generate_section> };

Extension rule is applicable to a certain value set if:

- base rule is applicable to this value set (actually for its corresponding subset);
- value set satisfies the select section of extending rule;
- the extending rule was not applied to this value set before.

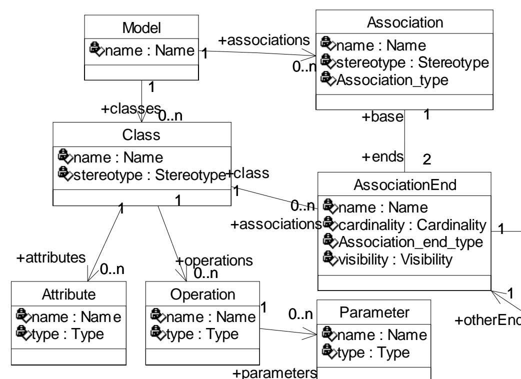
Execution of an extending rule is execution of its generation section. But unlike an ordinary rule it does not create a new transformation bond; instead it extends the bond created by application of the base rule with variables declared in the extension. It works similarly to inheritance in common programming languages: old attributes remain intact and new ones are added. Such bonds will be shown as results of calculation of navigation expressions that scan for applications of the base rule as well as applications of extending rule.

The concept of rule extension allows to structure the set of rules as well as the set of transformation bonds created by those rules. It allows easier development, modification and understanding of complex transformations.

Template is a special rule that begins with a key word ‘abstract’. Such rule is never applicable, no matter the select section. But it can be used as a base rule to create extensions that can be applied as normal rules. A template can extend another template but not an ordinary rule. Just as normal extensions, templates allow to structure transformation bonds and then use navigation expressions that query those bonds to define complex transformations.

8 Example of transformation definition

A simple transformation module with several rules is shown below. This transformation definition is meant to be used for transformation of a pair of models (‘source’ and ‘target’), both of which use UML class diagram metamodel. First model is a source data, and second is generated during the transformation. A simplified metamodel of UML class diagram used in this transformation definition is shown on the scheme:



Simplified metamodel of UML class diagram.

```
block example_transform {

rule class_mapping {
forall src from source/classes
make trgt in target/classes;
trgt/name=src/name;
}
```

This is module’s header and its first rule. The rule ‘class_mapping’ for each class of source model creates a class in the target model with the same name. Each application of this rule creates an instance of a transformation bond with name ‘class_mapping’ and attributes ‘src’ and ‘trgt’ that point to a class in source model and newly generated class correspondingly.

```
rule private_to_private {
forall a from source/classes
forall b from a/attributes
where b/visibility="private"
forall c from target/classes
forall d from rules/class_mapping
where (d.src=a) and (d.trg=c)
make e in c/attributes;
e/name=b/name;
e/type=b/type;
e/visibility="private";}
```

This rule copies private attributes from source to target model. The select section of this rule uses transformation bond created by application of previous rule. Because of this bond an attribute is copied exactly to the class that is generated by the class that contains this attribute, without the bond we would not know what class of target model to choose.

```
rule public_to_private {
forall a from source/classes
forall b from a/attributes
where b/visibility="public"
forall c from target/classes
forall d from rules/class_mapping
where (d.src=a) and (d.trgt=c)
make e in c/operations; // "get_elt()"
e/name="get_"+b/name;
e/type=b/type;
```

```
e/visibility="public";
make f in c/operations; // "set_elt(type)"
f/name="set_"+b/name;
f/type="void";
f/visibility="public";
make g in f/parameters;
g/name=make_unique_name(b/name);
g/type=b/type;
make h in c/attributes; // "private elt"
h/name=b/name;
h/type=b/type;
h/visibility="private";
}
```

This rule maps public attributes to private and creates corresponding 'get' and 'set' pairs of operations. A function "make_unique_name" that is used in this rule is a built-in function with a relatively simple functionality: it returns a string-name that is warranted to be unique in the scope of a model. Exact implementation of this function may differ in different transformation tools.

```
abstract rule all_inherited {
forall inherit_trgt from source/classes
forall inherit_src from source/classes;
}
rule directly_inherited:all_inherited {
forall assoc from inherit_trgt/associations
where ((assoc/stereotype="generalization") and
 (assoc/otherend=inherit_src))
}
rule indirectly_inherited:all_inherited {
forall r1 from rules/directly_inherited
where r1/inherit_trgt=inherit_trgt
forall r from rules/all_inherited
where ((r/inherit_trgt=r1/inherit_src) and
(r/inherit_src=inherit_src))
}
```

This set of a template and two rules defines the concept of "indirect inheritance" (if by inheritance we mean generalization association). The rules themselves do not modify the model, but they create transformation bonds that can be used by other rules. For example, here is a rule that generates a generalization association between a class and all its direct and indirect ancestors:

```
rule inheritance {
forall a from rules/indirectly_inherited
make b in a/inherit_src/associations;
b/cardinality="1";
make c in a/inherit_trgt/associations;
c/cardinality="1";
c/stereotype="generalization";
b/other_end=c;
c/other_end=b;
make assoc in target/associations;
include b in assoc/ends;
include c in assoc/ends;
b/base=assoc;
c/base=assoc;
}
} //example_transform
```

The last brace marks an end of the module "example_transform".

9 Conclusion.

MDA technology has a potential to become a new stage in evolution of software development toolkits and methods. But it is possible only if tools are created that are specially developed to support this technology and utilize its full potential. Existing tools that claim to support MDA are mostly minor modification of old tools that do not provide necessary functionality. One of the main problems that slow down appearance of new generation of tools is automated model transformation. Existing approaches from other areas of mathematics and informatics are inefficient for practical tasks of MDA-based development. That's why development of a new system and language is important.

This work offers a transformation definition language that is mostly meant to be used for automated transformation of UML models. During development of this language a special attention was paid to make it suitable for MDA tasks, and to make it easier to understand for a human. A prototype tool is being developed that supports this language and executes model transformations.

References:

[1] D. Chamberlin. XQuery: An XML query language. IBM systems journal, no 4, 2002.

[2] Common Warehouse Metamodel (CWM) Specification. OMG Documents, Feb. 2001. http://www.omg.org/cgi-bin/apps/doc?formal/03-03-02.pdf

[3] Krzysztof Czarnecki, Simon Helsen. Classification of Model Transformation Approaches. University of Waterloo, Canada, 2003.

[4] Keith Duddy, Anna Gerber. Declarative Transformation for Object-Oriented Models. Transformation of Knowledge, Information, and Data: Theory and Applications, 2003.

[5] M. Kuznetsov. Model Driven Architecture and Transformation of UML Models. In SYRCoDIS, pages 82-86. May 2004.

[6] Martin Flower. UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition. Addison Wesley, 2003.

[7] Tracy Gardner, Catherine Griffin A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. http://www.omg.org/docs/ad/03-08-02.pdf

[8] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, Andrew Wood. Transformation: The Missing Link of MDA. Proceedings 1st

International Conference on Graph Transformation (ICGT 2002), 2002.

[9] Anneke Kleppe, Jos Warmer, Wim Bast. MDA Explained. The Model Driven Architecture: Practice and Promise. Pearson Education, 2003.

[10] Jernej Kovse, Theo Härder. Generic XMI-Based UML Model Transformations. ZDNet UK Whitepapers, 2002.

[11] Meta-Object Facility (MOF) specification, version 1.4 . OMG Documents, Apr. 2002. http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf

[12] Joaquin Miller and Jishnu Mukerji(eds.). MDA Guide Version 1.0. OMG document, 2003. http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf

[13] Daniel Varro, Andras Pataricza. UML Action Semantics For Model Transformation Systems. Periodica Polytechnica, no. 3-4, 2003.

[14] XSL Transformations (XSLT) v1.0. W3C Recommendation, Nov. 1999. http://www.w3.org/TR/xslt