IADIS International Journal on Computer Science and Information Systems
Vol. 3, No. 1, pp. 29-43
ISSN: 1646-3692

TOWARDS MODEL TRANSFORMATION - A CASE STUDY OF GENERIC FRAMEWORK FOR TRACEABILITY IN OBJECT-ORIENTED DESIGNS

Anna Derezińska, Jacek Zawłocki *Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19,00-665 Warsaw, Poland*

A.Derezinska@ii.pw.edu.pl

ABSTRACT

Model transformation approach allows us to develop automatic and flexible solutions for the software evolution. Application of model transformation concepts was shown on a case study of a generic framework for traceability in object-oriented designs. Traceability analysis in the framework is based on discovering traceability relationships among model elements and identifying dependency areas for given initial elements. The analysis is controlled by a set of rules that should be easily modified according to the project notation, application domain, etc. Three transformations within the framework were considered: the input transformation of any model to the internal format, the traceability analysis generating a dependency area for a given model, and the output transformation of the resulting dependency area. They can be realized as model-to-model transformations with respect to their metamodels and in accordance to the independently specified transformation rules. The language and tools of the QVT standard proposed by the OMG were applied in the input and output transformations. In the remaining transformation, traceability rules were defined as automata with transitions labeled with conditions and actions.

KEYWORDS

model transformation, UML, MDA, QVT, traceability

1. INTRODUCTION

Model Driven Development (or Model Driven Engineering) is an idea promoting the shift from the code-oriented to model-oriented software production techniques. An important role is played by models and their transformations.

Model transformation allows defining clearly relationships between models. Performing model transformation requires understanding of the syntax and semantics of both the source and target [25]. Therefore, model transformation makes use of metamodeling techniques. Metamodeling is a common technique for defining precisely a class of models: the abstract syntax of models and the interrelationships between model elements.

Model Driven Architecture™ (MDA) [13,21] was intended as the realization of the model engineering principles around the set of the Object Management Group (OMG) standards. One of them is QVT (Query/View/Transformation) [22] devoted to model transformation and based on metamodeling concepts.

Many approaches to model-to-model transformation have been proposed [5,6], but there is a lack of sufficient experience in their practical application. Transformation capabilities of modeling tools available on the market are growing up, but are still premature. An important question is how well QVT and other MDA-based solutions fit for different kinds of transformations [3].

In this paper, we show a practical application of a model transformation in the development of a generic software system. The system is a framework for traceability of object-oriented projects [10,31]. A distinguishing feature of the generic framework is its flexibility. One of the basic concepts is separation of application logic and its implementation. The same postulate is true for transformation i.e., separation of transformation rules encapsulating the internal logic and the transformation execution. Advantages but also drawbacks of the transformation-based solutions are presented.

Traceability refers to relations, which are defined among different artifacts of a software development process. It is beneficial in requirements verification and evolution, models and code development and maintenance, tests generation and management [11,17,18,26-28].

Traceability issues addressed in our framework are aimed at object-oriented projects. However, those projects can be described at different abstraction levels e.g., consisting of analytical or implementation classes, but also general concepts from a given domain. The framework is based on the original ideas of dependency areas developed by one of the authors [7-9], but it is not limited to the traceability strategy proposed for UML. The framework can help solving different problems related to:

- identification of dependencies and inconsistencies in a project,
- impact analysis of changes within a project,
- support for model understanding, including reverse engineered models and legacy systems,
- creating documentation of a project,
- support for model and code instrumentation.

The rest of the paper is organized as follows. Section 2 explains basic concepts of model transformation. Next section summarizes briefly main features of the generic framework for traceability in object-oriented designs. Section 4 describes selected transformation issues within the framework. Final remarks conclude the paper.

2. MODEL TRANSFORMATION

Model transformation deals with manipulation of different system abstractions expressed by models. A variety of software development artifacts can be a subject of transformations. Survey and classification of model transformation approaches can be found in [5,6].

Model transformation is a key part of MDA that serves as a conceptual framework for an approach to model-driven development. For creating transformations we can apply the OMG specification MOF QVT [22]. The QVT Relations language is a declarative language for model-to-model transformation. It can be used for writing own transformations or adopting some existing transformations to our needs.

In order to make a transformation between different notations of a model, or between models of different abstraction, we need a common core of basic concepts. In the QVT approach this was achieved using the meta-modeling layers developed by the OMG specifications (Fig. 1). Layer M1 includes models describing systems in different application domains. Model specification languages (e.g., UML) are described by their metamodels (M2 layer). It was provided a language for defining metamodels i.e. a meta-metamodel level, called MOF (Meta Object Facility). QVT supports a transformation of the MOF metamodel and, therefore, a transformation of any models defined using MOF.

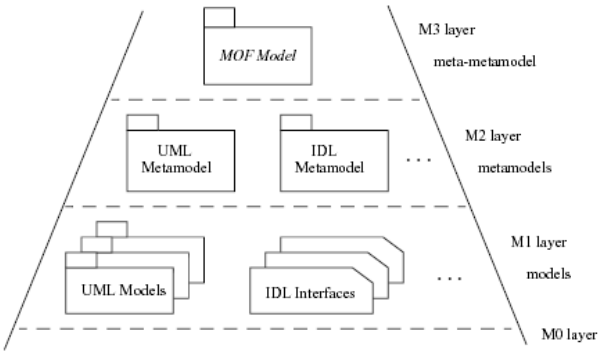

Figure 1. The role of the MOF in the metamodel hierarchy

A concept of model transformation realized in QVT is illustrated in Fig. 2. There are two groups of models: source models and target models. These models are instances of given metamodels. Metamodels can be in general different but both conform to the meta-metamodel MOF, as they are instances of this meta-metamodel. A transformation is defined with respect to the metamodels. There is a set of rules defining the transformation process. The transformation engine converts a source model to the appropriate target model according to the transformation rules.

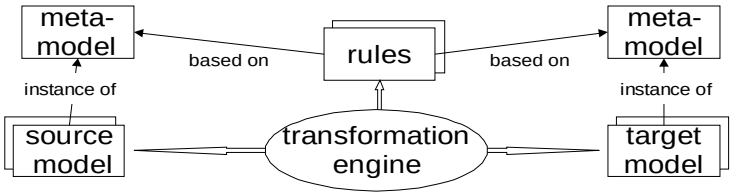

Figure 2. Idea of model transformation in QVT

The QVT standard is a general purpose language for model-to-model transformations and, therefore, can be applied in different situations. The MOF QVT specification can be used for transformation on different levels of abstraction, for forward, but also reverse transformation, enabling possibility of automatic synchronizations and re-refinements [12]. In [19] the author points out on the possibility of usage of QVT for description of a generic model of Quality-Driven Software Architecture (QAMT).

Although QVT was developed by OMG as one of specifications within the MDA approach, there are many other solutions used for this purposes [1,3]. Czarnecki and Halsed [6] provided an analysis of different transformation approaches, including QVT proposals. Different languages were proposed for model transformation purposes: QVT [22], ATL [15], Viatra2 [2], EWL [16] and others (see survey in [5]). Transformation rules were successfully described in OCL, but also in other formal notations (e.g., Object-Z, B, Maude).

Similarly to program transformation we can distinguish two general categories of model transformation: *translation* and *rephrasing* [24], also called as *mapping* and *update* [6]. In the first case, a source model can be transformed into a target model of a different language. There is usual a direct correspondence between the sub-sets of elements from both models. In the later case, a model is changed in some way producing a new target model. The application of both approaches will be discussed in the paper.

3. GENERIC FRAMEWORK FOR TRACEABILITY IN OBJECT_ORIENTED PROJECTS

3.1 Traceability Concepts

A notion of traceability can be in general understood as a directed relationship between source and target entities. The relations can be specified for any artifacts created within a software development process (from requirements, trough models and test cases, to the code), or more specifically for selected elements of the process. An overview of traceability issues in the software development can be found in [18].

The framework discussed in this paper deals not with the whole software life cycle, but the traceability is limited to the relations within an object-oriented design. It is based on the traceability concepts of *Dependency Areas* [7-9]. In general, it considers an object-oriented project described at different levels of abstraction. For a given project and an initial element it extracts a subset of the mostly related elements and identifies relations between them.

This extraction process can be ruled by different traceability strategies. They determine, for example, kinds of relations between elements that are considered as "the mostly related". A traceability strategy is defined by a set of traceability rules.

The concepts of Dependency Areas were developed in the context of UML language, especially taking into account incomplete models. However the idea can be applied for different subsets of UML, or different models using object-oriented paradigm.

Traceability analysis is defined as a transformation that takes as an input a vector consisting of the following elements: *Project - P*, *InitialElement - IE*□ *P* and *TraceabilityStrategy - R.* Result of the transformation is *DependencyArea - DA* □ *P*.

$$\{P, IE, R\} \rightarrow \{DA\}$$

It should be noted that there are other research issues dealing with traceability and model transformation that should not be confused with the approach discussed in this paper. Traceability is supported in QVT where instances of trace classes store the record of transformations. However the presented approach is not about traceability within transformation process, but vice versa, about usage of transformations in development of a framework for recognition and elicitation of traceability relations.

In [4] transformations between models, and between models and code are realized using QVT. Artifacts maybe transformed in other artifacts, using some kind of transformation process available (fully automatic, assisted or manual) and traces are maintained when a transformation occurs. Keeping the coherence between the artifacts of the system also after transformation activities is the main goal of the presented framework.

The solution described in [27] is also about traceability within Model Driven Development (MDD), but not about application of MDD. Generated traces provide information that can be further used in transformation between models - the problem is, therefore, opposite to presented here. In [28] the set of services: trace model management, trace creation, trace use and trace monitoring are discussed. The services could support any kinds of artifacts and relations in a heterogeneous MDD environment. However the solution has not yet been prototyped or evaluated.

3.2 Framework Requirements

The framework is devoted to traceability in software designs. Its goal is discovering traceability relationships in a given object-oriented model according to a given traceability strategy in an automatic way. After analysis of the software development process in small companies, the following needs were recognized:

- the framework should be able to cope with UML notation,
- the framework could be extended also for other notations and support traceability in projects specified with these notations,
- the framework should be adaptable for new UML meta-models
- the logic of traceability process should be defined using a separate layer of user interface.

The framework supports traceability in object-oriented designs identifying dependency areas. The framework was intended to be generic and highly flexible. The configurability of the framework is based on multi-tier architecture, state-machine theory, scripting languages provided to end-users and plug-in mechanisms. An object-oriented model in any notation

can be accepted by the framework. It requires only a pre-processing realized by an appropriate input plug-in. It converts any model to an internal form (so-called *Project* notation).

Traceability analysis is performed on a Project model according to a given subset of traceability rules. Each rule is defined by a finite state automaton. Automata are interpreted by an engine of the framework, so-called traceability analyzer. More details about the framework architecture can be found in [10,31].

The high flexibility of the framework implies different activities that require knowledge and different levels of interference into the framework. Therefore, we can distinguish three roles of users. The actors of the framework with their basic use cases are shown in Fig. 3.

Application of the generic framework to practical purposes requires its adaptation. It is a role of a *plug-in developer* and a *traceability process modeler* that prepare the platform to be used by a *model designer*. The following analytical and technical tasks should be realized:

- problem recognition and its analysis concerning possibility of solutions with the traceability analyzer,
- selection of model notation for description of the problem,
- preparation of conversion from the given model notation to the notation accepted by the traceability analyzer (Project format),
- design of a traceability logic in dependence of the considered problem (selection among prepared traceability strategies),
- preparation of conversion of a dependency area from the form given by the traceability analyzer to a suitable output form (if required).

Figure 3. Actors and use cases of the framework

3.3 Framework Structure and Processes

The general process realized within the framework is shown in Fig. 4. In order to satisfy the flexibility requirements of the framework we used model transformation approaches. In the process supported by the framework we can distinguish three main points, where model transformation can be applied:

1. *input transformation*: from an object-oriented model to the model in the internal (Project) notation,

2. *traceability transformation*: from a Project model with a given initial element to the resulting Dependency Area,
3. *output transformation*: from a Dependency Area to a resulting model in a desired notation.

In general, the input and output transformations can be classified as *model translations (mapping)*; more precisely *model migrations,* because a model is transformed to another one at the same level of abstraction [24]. The later case, i.e., the traceability transformation, is an example of *model rephrasing (update)*, and within this category a kind of *model adaptation.* The result of traceability analysis can be described by the same model notation, but the model is changed in order to reveal new features.

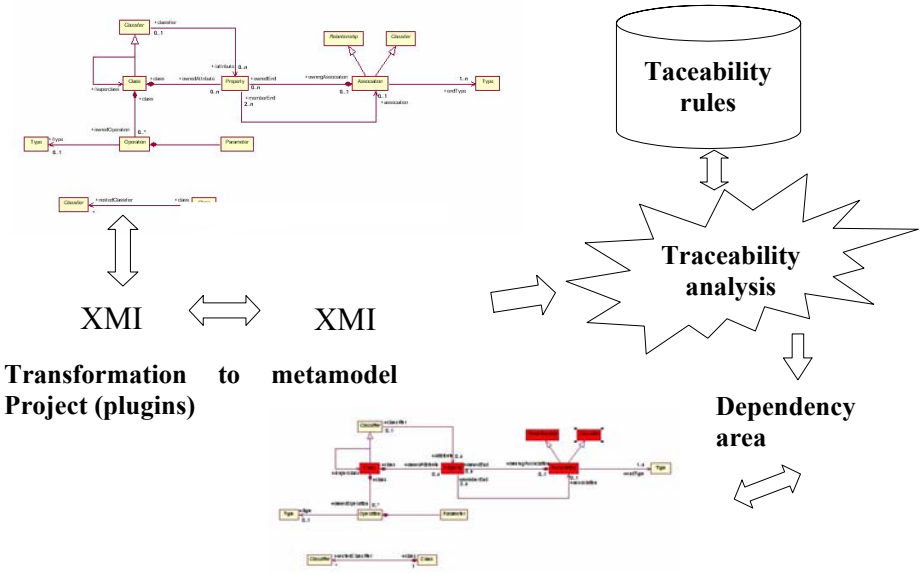

Figure 4. Realization of traceability process

An exemplary instance of the framework was implemented. It handles selected parts of the UML metamodel and its traceability. Models can be serialized as Extensible Markup Language (XML) using the XML Metadata Interchange (XMI) [23].

Input transformation was realized using QVT approach and existing, supporting it tools (Sec. 4.2). Output transformation (Sec. 4.3) is analogous to the input one. The main, traceability transformation follows the described concepts, but it was performed by the dedicated traceability analyzer (Sec. 4.1). It transformed models according to the traceability rules defined as the specialized automata. Therefore, the principle of separation between traceability logic and its execution was preserved.

The architecture of the framework consists of four layers. The layer of *Rule Processing* is responsible for traceability transformation (Sec. 4.1). It uses project and traceability rules delivered by other layers. The *Input/Output* layer comprises three components: one - an interface for converting a project to a form suitable to the *RuleProcessing* layer, second - an

interface for serializing of generated dependency area. The third component delivers traceability rules from a given XML file. Next layer is a plug-in layer. It includes implementation of the interfaces from the *Input/Output* layer. The final, *Data* layer comprises XML files with traceability rules and documents with input project and output results.

4. APPLICATION OF MODEL TRANSFORMATION IN THE FRAMEWORK

This section describes transformation solutions used in the framework. All of them are consistent with the concepts illustrated in Fig. 2.

4.1 Transformation Process of Traceability Analyzer

Internal form of an object-oriented project is described by the metamodel Project. Traceability rules are designed to operate on any model consistent with the Project metamodel. It is a general form to which models of specific types can be transformed.

A project (class *Project*) consists of many elements (class *ProjectElement*) (Fig. 5). Any element of a project can have any number of annotations (class *LinkAnnotation*) referring to traceability process. Annotations define a set of other elements related in the project, specifying their identifiers, types or names.

Such a project can be understood by the executor of traceability rules. According to the interpretation of MDA transformation, any instance of metamodel Project should be transformed to a dependency model. This model is an instance of the metamodel of Dependency Area (Fig.6). It is a resulting metamodel of the traceability process.

A dependency area aggregates a set of area members. Each member is a specialization of an element of the Project. An area member can have a number of links (class *Link*) to other elements of the dependency area. Links can have their priorities and types, defined according to performed traceability rules.

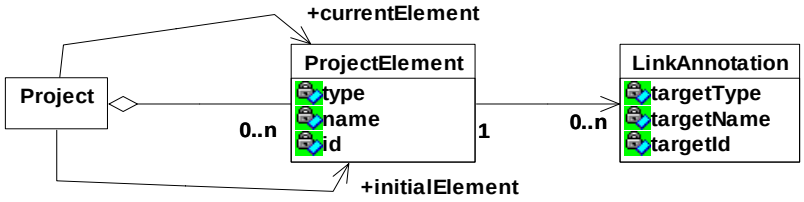

Figure 5. The core of the Project metamodel

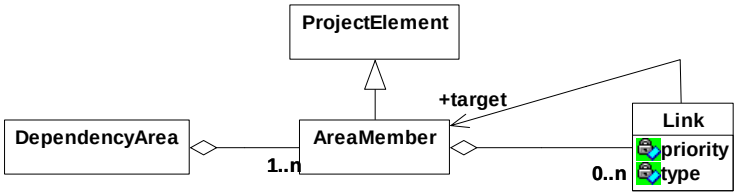

Figure 6. The core of the Dependency Area metamodel

Traceability rules are specified using automata approach. Each rule is a separate, finite state automaton. A status of a rule execution is stored in the nodes of automata. Transitions between nodes are annotated with actions and conditions. An action can be performed in a given state only if the appropriate condition is satisfied. Actions and conditions are specified in a scripting language. In the implementation we used JavaScript language. The detailed description of the syntax of traceability rules is beyond the scope of this paper [31].

The automata approach is very flexible allowing specification of different transformation rules, including different traceability policies. On the other hand, the transformation can be realized by a simple rule executor, because the entire logic is stored in the rules. The rule executor executes a set of rules for any project element that was assigned to a dependency area. All rules are ordered according to their priorities and executed in the defined order. If two or more rules have the same priority the order of their execution is random. Elements assigned to a dependency area are considered by the executor in the order of their addition. The first one is the initial element indicated in the input project.

Before executing a single rule, its precondition is checked. If it is satisfied, the initial node of the rule is considered. In case a node has more than one outgoing transitions, they are ordered according to edge priorities. If a condition of a selected transition is satisfied, the transition is followed and its action performed. All nodes of the rule accessible from its initial node are visited during the rule execution.

In the result of the transformation process, the whole set of rules is executed for any project element assigned to the dependency area. Any element is added only once to the resulting dependency area. If an action specifies assignment of an already existing element, only additional references between elements are added in the output dependency model.

4.2 Transformation Process of Input Models

According to the assumptions of the framework, an input model can be specified in any form that can be transformed to an internal form of the Project metamodel. The transformation of the input model can be realized by an input plug-in. The idea will be explained on an example of a subset of UML.

General approach corresponds to that shown in Fig. 2. An input UML model can comprise any elements, but only elements interpreted by transformation rules in the input transformation process will be further handled by the traceability analyzer. We assumed, that an initial element is denoted by the stereotype «starting» associated with one element of a UML input model. Therefore transformation rules have to take into account notion of stereotypes.

Transformation rules are defined as a set of declarations satisfying the requirements of the QVT specification. The set of rules describes mapping of elements of a project defined according to one metamodel, to elements of a project from another metamodel. Transformation rules from UML to Project are straightforward. Figure 7 illustrates the idea of exemplary rules. On the left hand side input elements of UML are given. The set of rules takes these elements as their inputs. Results of the rules are instances of classes *ProjectElement* and *LinkAnnotation* shown on the right hand side. These instances are connected with appropriate references.

Transformation rules should be described accordingly to a used transformation tool. Detailed syntax of the rules accepted by a tool (MdaTranfs [29]) used in the implementation can be found in [30]. The rules are specified in the XML language. An exemplary rule is shown in Fig. 8. It transforms metaclass *Attribute* from UML metamodel to class *ProjectElement* from Project metamodel.

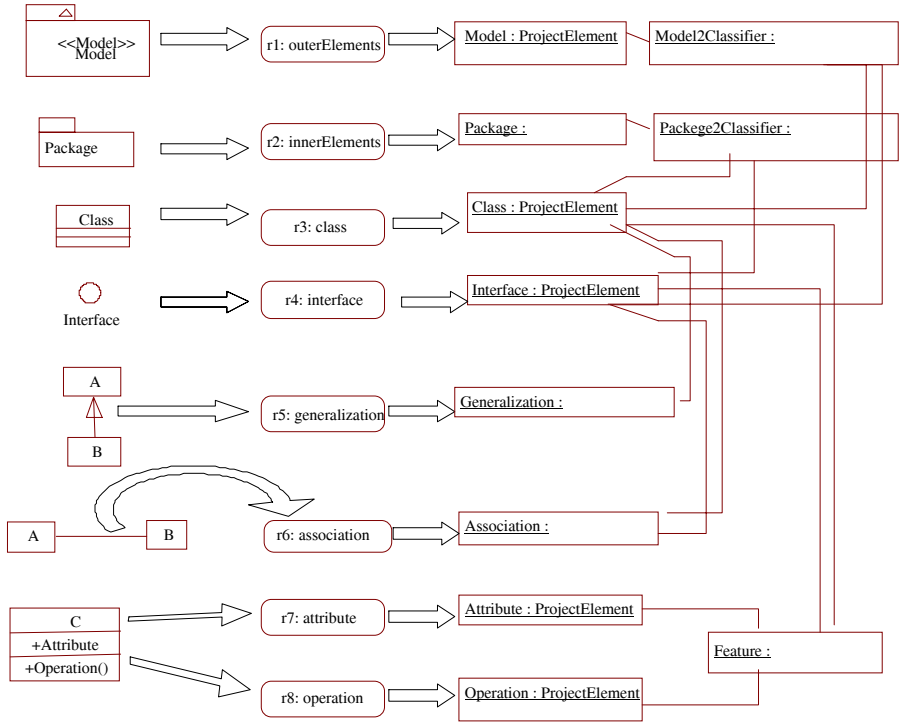

Figure 7. Examples of transformation rules from a UML model to Project elements

```
<rule name="attribute">
        <domain model="uml14" varName="c1" type="Core.Attribute">
                <primitiveProperty name="name" varName="n" type="String"/>
                <collProperty name="stereotypes" varName="s" type="String"/>
        </domain>
        <domain model="trace" varName="c2" type="pw.ii.trace.plugin.xmi.project.model.ProjectElement">
                <primitiveProperty name="name" varName="n" type="String"/>
                <primitiveProperty name="id" initValue="c1.refMofId()" type="String"/>
                <primitiveProperty name="starting" type="String"
        initValue="(s.contains('starting')?'true':'false')"/>
                <primitiveProperty name="type" type="String"
        initValue="c1.getClass().getInterfaces()[0].getName()"/>
        </domain>
</rule>
```

Figure 8. A specification of a transformation rule from UML to Project - transformation of an attribute

The schema of transformation process of an input model is shown in Fig. 9. It consists of the following steps:

1) A metamodel of the Project is prepared using a CASE tool and exported in the XMI format.

2) This metamodel is transformed from the UML format to the standard MOF format.

3) A metamodel of UML in the MOF format is delivered.

4) Metamodels of UML and Project are converted from the XMI to JMI standard.

5) Transformation rules are specified in the XML form accepted by the transformation tool.

6) A UML model, to be analyzed, is prepared in a CASE tool and exported to the XMI format.

7) The UML model is converted from the XMI format to the JMI standard.

8) The input UML model is transformed to its corresponding Project model, using UML and Project metamodels, and the appropriate transformation rules.

It should be noted that steps from 1 to 5 should be performed only once for a given input model notation. They are the tasks of a plug-in developer. Only steps 6-8 are performed by a model designer each time a new model is analyzed. Step 6 is realized manually - it is a proper design activity, whereas steps 7 and 8 are completed automatically using the previously prepared plug-in.

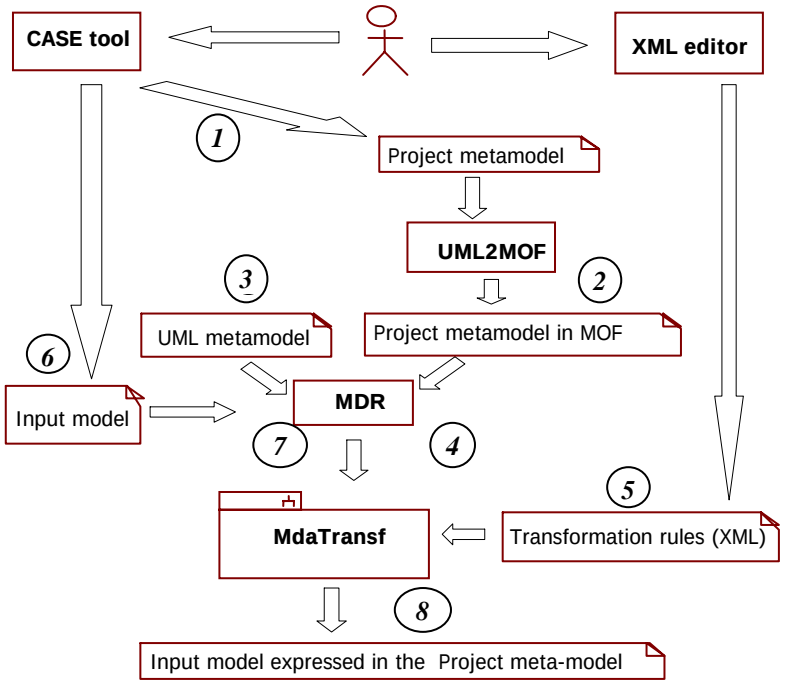

Figure 9. Realization of the input transformation process

A result of the transformation process is the input model described as a Project model in the XMI format, suitable for the traceability analysis.

In an exemplary implementation of the input plug-in we used the following tools supporting the transformation process:

- UML2MOF - a tool for model conversion from UML to MOF,
- MDR [20] - a library implementing the JMI standard [14] that describes generation of interfaces for accessing model elements based on the metamodels given in the XMI,
- MdaTranfs [29] - a tool for model transformation based on the QVT approach.

Generation of interfaces (steps 4 and 7) is realized before the models can be read by the transformation tool. The JMI standard defines also the set of operations that can be performed on the models, e.g., searching according to a type, getting a value, modification of values, etc.

4.3 Transformation Process of Output Models

The output transformation should present the results delivered by the traceability analyzer in a form suitable for a user. The internal form of a dependency area model should be converted into an equivalent, legible form. There can be, of course, many different transformations that satisfy different user demands. Similarly, as for the input transformation, we discuss the exemplary solution for UML.

The output transformation process (Fig. 10) can have a similar structure as that shown for the input process. Three types of input data should be prepared for a given output nota-

tion. If the notation of the output result is UML, three types of input data are metamodel of Dependency Area, metamodel of UML and output transformation rules (in XML). Instead of a CASE tool, as in the input transformation (Fig. 9), the traceability analyzer can be found in the output process. A dependency area generated by the traceability analyzer is the fourth input of the process.

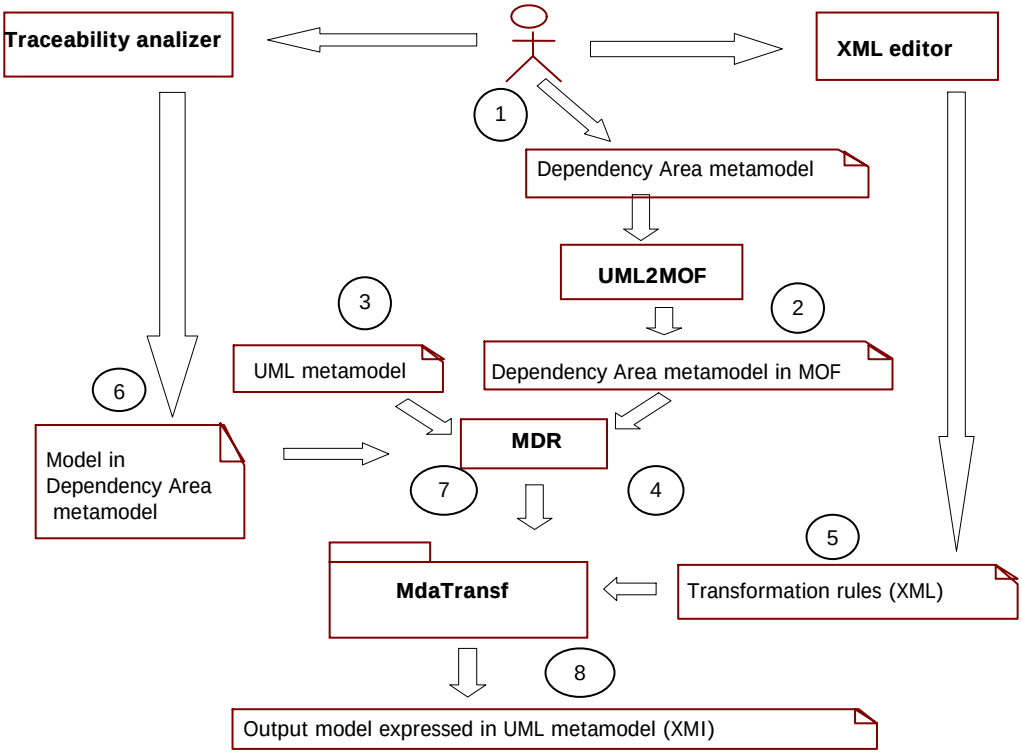

Figure 10. Realization of the output transformation process

Using the same tools as shown for the input transformation process, we can automatically obtain the dependency area as a UML model. In this case transformation rules specify transformation from Project metamodel to UML metamodel. They are written in XML according to the syntax [30], similarly as in the input transformation process.

Once, having prepared the appropriate metamodels and transformation rules, the whole output transformation process is transparent to a user. Although a plug-in developer can modify the metamodels and/or transformation rules and, therefore, adjust the process on demand.

5. CONCLUSIONS

We have shown how model transformation technology was used in the generic framework. The framework supports traceability in object-oriented designs. Model transformations

assisted to attain the important advantages of the framework i.e., its flexibility, independence of the notation of an input model, separation of the traceability logic from the execution engine, and possibility of the simple evolution of the traceability strategy. However, in order to use this technology, it was necessary to provide specifications of the appropriate metamodels and prepare definitions of required transformation rules. In both cases of languages used for the transformation descriptions, the QVT and the automata-based approach, preparation of the transformation rules was very laborious. It was the price of the achieved universal solution.

Further development of the framework should deal with plug-ins for other UML subsets or specialized meta-models. Evaluation of different traceability strategies will require more experiments with different sets of traceability rules. In the further evolution of the framework we can benefit from the model transformations applied in it.

REFERENCES

1. Arujo, J. et al., 2002. Integration and transformation of UML models, *Proceedings of ECCOP Workshops*. LNCS Vol. 254, Springer Berlin Heidelberg, pp.184-191.
2. Balogh A., Varro D., 2006. Advanced model transformation language constructs in the VIATRA2 framework. *In Proceedings of the ACM symposium on Applied Computing (SAC'06)*. New York, NY, USA, pp. 1280-1287.
3. Buttner, F., Bauerdic, H., 2006. Realizing UML Model Transformations with USE. *Proceedings of UML/MoDels Workshop on OCL*. Genova, Italy, pp. 96-110.
4. Costa M., da Silva A. R., 2007. RT-MDD Framework - A Practical Approach. *In Proceedings of ECMDA Traceability Workshop*. Haifa, Israel, pp. 17-26.
5. Czarnecki, K., Helsen, S., 2006. Feature-based Survey of Model Transformation Approaches. *IBM System Journal*, Vol. 45, No 3, pp. 621-645.
6. Czarnecki, K., Helsen, S., 2003. Classification of Model Transformation Approaches. *In Proceedings of the 2nd Workshop on Generative Techniques in the Context of Model Driven Architecture (co-located with OOPSLA'03)*
7. Derezińska, A., Bluemke, I., 2005. A Framework for Identification of Dependency Areas in UML Designs, *Proc. of IASTED Conf. on Software Engineering and Application*, SEA'05, Phoenix, Arizona, USA, Acta Press, pp. 177-182.
8. Derezińska, A., 2006. Specification of Dependency Areas in UML Designs, *Annales UMCS Informatica* AI 4, Vol. 4, pp. 72-85.
9. Derezińska, A., 2004. Reasoning about Traceability in Imperfect UML Projects. *Foundations of Computing and Decision Sciences*.Vol. 29, No. 1, pp. 43-58.
10. Derezińska, A., Zawłocki, J., 2007. Generic Framework for Automatic Traceability in Object-Oriented Designs. *Annals of Gdansk University of Technology Faculty of ETI*, No 5, pp. 291-298 (in polish).
11. Egyed, A., 2004. Consistent Adaptation and Evolution of Class Diagrams during Refinement, *Proceedings of 7th Inter. Conf. on Fundamental Approaches to Software Engineering* (FASE). Barcelona, Spain, pp. 37-53.
12. Fondement F., Silaghi R., 2004. Defining Model Driven Engineering Processes, *Proceedings of 3rd Workshop in Software Model Engineering* (Wisme'04). Lisbon, Portugal.
13. Frankel, D. S., 2003. *Model Driven Architecture: Appling MDA to enterprise computing*, Wiley Press, Hoboken, NJ, USA.

14. JMI standard, http://java.sun.com/products/jmi/
15. Jouault F., Kurtev I., 2005. Transforming Models with the ATL. In *Proccedings of the Model Transformation in Practice Workshop at MoDELS 05*. Montego Bay, Jamaica, LNCS Vol. 3844, pp. 128-138.
16. Kolovos D.S. at al., 2007.Update Transformations in the Small with the Epsilon Wizard Language., *In Journal of Object Technology.* Vol. 6, No. 9, Special Issue TOOLS EUROPE Oct. 07, pp.53-69.
17. Letelier, P., 2002. A Framework for Requirements Traceability in UML-based Projects. *Proceedings of 1st Int. Workshop on Traceability in Emerging Forms of Software Engineering*. (*co-located with IEEE Conf. on ASE),* Sept. 28, Edinburg, UK.
18. Maeder P. et al. 2006. Traceability for managing evolutionary change. *In Proceedings of 15th SEDE (ICSA).* Los Angeles, USA, pp.1-8.
19. Matinlassi M., 2005. Quality-Driven Software Architecture Model Transformation. In Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). Pittsburgh, Pennsylvania, USA, pp. 199-200.
20. MDR library, http://mdr.netbeans.org/
21. Object Management Group, 2003. *MDA Guide*, Ver. 1.0.1, OMG Document omg/2003-06-01.
22. Object Management Group, 2005. *MOF QVT Final Adopted Specification*, OMG Specifications ptc/05-11-01.
23. Object Management Group, 2005. *MOF 2.0/XMI Mapping Specification*, Ver. 2.1, OMG Document formal/05-09-01.
24. Sendall S. at al., 2004. Understanding Model Transformation by Classification and Formalization, Proceedings of Workshop on Software Transformation Systems (part of 3rd International Conference on Generative Programming and Component Engineering). Vancouver, Canada.
25. Sendall S., Kozaczynski W., 2003. Model Transformation - Heart and Soul of Model-Driven Development. *In IEEE Software*, Vol. 20, No. 5, pp. 42-45.
26. Spanoudakis, G. et al., 2004. Rule-based Generation of Requirements Traceability Relations, *Journal on Systems and Software*, Vol. 72, No. 2, pp. 105–127.
27. Vanhooff B. et al., 2007. Traceability as Input for Model Transformations. *In Proceedings of ECMDA Traceability Workshop.* Haifa, Israel, pp. 37-46.
28. Walderhaug S., et al. 2006. Towards a Generic Solution for Traceability in MDD. *In Proceedings of ECMDA Traceability Workshop.* Bilbao, Spain.
29. West Team, *MDA - Transf User Guide*, http://www.lifl.fr/west/modtransf/
30. West Team, *MDATrans Syntax*, http://www.lifl.fr/~dumoulin/modTransf/doc/
31. Zawłocki, J., 2006. *A Framework for Traceability Process in Object-Oriented Models*. Master Thesis, Institute of Computer Science, Warsaw University of Technology (in polish)