Model Management Based on a Visual Transformation Language

Michał Śmielek\textsuperscript{1,2}, Andzej Kardas\textsuperscript{1}

Warsaw University of Technology\textsuperscript{1}, and Inovide S.A.\textsuperscript{2}, Warsaw, Poland
smielek@iim.pw.edu.pl

Abstract. Modeling becomes the most prominent method of dealing with software complexity. Models for software systems can be created on different levels of abstraction and in different phases of the software development process. In order to manage these various models effectively we need to apply methods for keeping them coherent and traceable. The current paper proposes a method for defining and executing transformations between models written in the currently most popular modeling language (UML). We propose a general purpose transformation language that allows for specifying transformation mappings between different models. This language has a visual (MOF-based) notation and allows for defining transformation templates and rules. With this language, the software developers are capable of determining traceability links between individual model elements. Moreover, the transformations can be executed automatically to high extent, with the use of a dedicated tool that extends a standard UML CASE tool.

1 Introduction

Complexity of contemporary software systems is constantly growing. In order to deal with this complexity effectively, appropriate abstraction techniques are needed. Creating models of software, slowly becomes the most prominent method for dealing with this complexity and realizing abstraction. Models allow for managing the structure and dynamics of even the most complex systems on different levels of abstraction, making it possible to reveal only the amount of detail that is comprehensible for the model readers.

Currently, the most widely used software development methodologies are based on the object oriented paradigm. After an initial boom in the area of software modeling methodologies based on object orientation, current tendency is to unify different approaches. The most prominent example in this area is the Unified Modeling Language (UML) \cite{OMG}. UML is already a standard used by vast majority of companies in the software industry and managed by an independent standardization body - the Object Management Group (OMG). UML in its current version 2.0 offers thirteen diagram types that enable showing various aspects of software and associated problem (business) domains on different levels of abstraction. Development of the UML standard goes in the direction of adding precise semantics to its diagram elements. Defining precise semantics
allows for bringing closer the idea of “visual programming”, where coding would mean drawing diagrams instead of writing textual code.

The most significant problem in using UML is to organize diagrams into models that would form a clear path from user requirements, through architectural design to detailed design and code. Taking this into account, the OMG has introduced a new modeling framework, called Model Driven Architecture (MDA) [2, 3]. MDA is based on the idea of model transformation. According to this idea, models close to the business domain (Computation Independent Models) would get transformed into abstract architectural models of the system, independent of the programming platform (Platform Independent Models). These models could then be transformed into models dependent on the software platform (Platform-Specific Models), and then directly translated into working code. Such a clear path, from visual, model based requirements, through design models to code could be partially automated, thus giving faster development cycles (see eg. [4], [5]).

In order to automate model transformation we need a transformation language that would allow for precise mapping of one type of models into another type of models [6]. An appropriate request for proposal has been issued by the OMG [7]. In reply to this proposal, several transformation languages have already been proposed (see eg. [8], [9], [10] and [11] for summary). However, most of the propositions concentrate on PIM to PSM mappings. In the current paper we propose a transformation language that is suitable also for mappings on the CIM level. This language is based on visual notation that allows for easier definition of model mappings. The paper also presents a tool that verifies applicability of the proposed language to transforming models stored in general CASE tool repositories.

2 Model transformations based on UML

The key concept of MDA is the definition of model transformation [12]. Having a standard modelling language, as UML, we have to define a way to translate one type of model into another type of model. This method would have to transform individual elements and relationships (like: classes, use cases, associations, messages, etc.) in one of the models into separate elements and relationships in the other model. We need to define this method and execute it on the source model in order to obtain automatically (or semi-automatically) the target model.

The method to be applied when transforming models usually depends on the needs of the model creators. For this reason, we need to develop a language that would allow defining different methods for model transformation. This language would offer capabilities to specify appropriate rules for mapping source elements into target elements. One of the method for specifying these rules is to define appropriate UML profiles (see eg. [13]). Another method is to define a meta-model of transformation in terms of OMG’s MOF [14] specification. This approach would be compatible with MOF-based definition of UML itself [1].
The MOF approach gives us the advantage of creating a formal specification for our transformation language by using this standard language-definition language (meta-language). In the meta-modeling sense, this specification is on the “meta” level, which is illustrated on Figure 1. With MOF, we can define an abstract language that possesses all the needed syntactic and semantic well-formedness rules. By adding to this abstract language also certain concrete syntactic notations, we receive a complete concrete, visual transformation language (VTL). This concrete language can be used by the model developers to determine model transformation rules. These rules can be used to perform actual transformations on the UML modeling level.

In the following two sections we shall present the proposed visual transformation language (VTL). We define the language’s abstract syntax by defining its MOF-compliant metamodel. We then specify the concrete syntax by giving an example of the proposed visual notation.

### 3 Visual transformation language - abstract syntax

The source and target elements of our transformations represent appropriate elements found in UML. We normally want to transform classes, interfaces, use cases, actors, components, etc. Such elements can be connected through various relationships: associations, dependencies, generalizations, etc. In order to perform a transformation on such a model we have to find an appropriate pattern, composed of source elements and relationships between them. Having such a pattern we can apply certain transformation rules. These rules determine the actual mapping from source elements and relationships into target elements and relationships. Our transformation language should, thus, be capable of defining templates for patterns in the source model, rules that describe mapping of elements consistent with these templates, and templates for relationships in the target model.

The above requirements for our language are illustrated on Figure 2. The language would be used to define transformation specifications. This model would contain source templates, transformation rules and target templates. The transformation model can be used to perform actual transformations (transformation instances). If one of the source template matches a fragment in the source model,
appropriate transformation rules are applied in order to obtain a transformed fragment in the target model. This target fragment is consistent with an appropriate target template.

Every transformation defined in our language, in order to be complete, has to include the following information:

- definition of the source elements of the transformation
- definition of the target elements of the transformation
- mapping rules that control the translation of the source model elements into the target model elements

This is reflected in the metamodel of our transformation language, shown on Figure 3. Every Transformation is composed of several source and target ComplexElementTemplates and several ComplexTransformationRules. It can be noted that all these elements of our metamodel are derived from the Package metaclass found in the UML specification (in the Kernel package) [1]. Basic elements of every Transformation are TransformationElements that reflect elements in the source and target models of a transformation instance. These TransformationElements participate in SimpleTransformationRules and SimpleElementTemplates. Every simple rule is composed of an appropriate link (TransformationLink or TemplateLink) that connects two TransformationElements.

Figure 4 shows important details of individual elements in our metamodel. Every TransformationElement has its type. This type is defined with the ElementTypes enumeration, which can have values reflecting various metaclasses found in the UML metamodel (like: Class, UseCase, Interface and so on). The transformationSide can be either source or target. This determines, whether the element can participate in source templates or in target templates. An important attribute of the TransformationElement metaclass is the stereotypeName. The stereotype allows for performing more fine-grained transformations than only based on element types. By setting the value of this attribute in a transformation element,
the transformation developer can determine differences in transforming model elements with different stereotypes. Another important meta-attribute of TransformationElement is packageName. The value of this attribute specifies the package in the source model where the element should be sought for or the package in the target model where the element should be placed in.

The remaining two meta-attributes of TransformationElement define the type of name conversion and member conversion respectively. Name conversion is very important, as it determines how the name of the current element will be changed in the element on the other side of an appropriate TransformationLink. One possibility is just to copy the name. Another two possibilities include adding a prefix or a postfix. The resulting name can be also set by the developer “by hand”. Member conversion for a transformation element specifies how the elements members (like attributes, parts or operations) will be converted in the other element. Here we will mention two possibilities: we can copy all the members or we can suppress copying (no members copied).

Name and member conversions are performed on elements (only between a source and a target element) connected with TransformationLinks. It can be noted that these links are directed. The direction determines whether this particular transformation can be performed between source and target only or in both directions. We can also observe, that the TransformationLink has the same
conversion attributes as TransformationElement. This is due to the fact that a single element can be converted to several other elements. In such situation each TransformationLink coming from a source element can add its own control over conversion in addition to standard conversion specified inside this element.

For the definition of transformation to be complete, we also need to define the TemplateLink. This connector determines relationships between elements in the source or target templates. TemplateLinks can have types that reflect metaclasses of the UML specification that derive from the Relationship metaclass. Thus, we can have TemplateLinks typed e.g. as Dependencies, Associations or Generalizations. TemplateLinks can connect only two TransformationElements on the same transformationSide. This allows for defining templates for the source or target models.

4 Visual transformation language - concrete syntax

We shall describe the concrete syntax of our Visual Transformation Language by using an example shown on Figures 5-7. These diagrams contain a VTL model (in consistence with what we said in section 2) of a specific transformation, compliant with the meta-model described in the previous section.

As it can be noted, VTL TransformationElements are denoted with a symbol identical to the UML’s class symbol. Every such symbol is named (as being derived from a UML NamedElement, see [1], Fig. 4 on p. 24). All the meta-attributes of TransformationElements are simply represented as attributes of transformation model elements with assigned specific values. TemplateLinks are denoted with appropriate connectors whose notation is derived from appropriate UML’s relationships (see Fig. 5 and 6). TransformationLinks are denoted as associations with an added constraint that reflects values of the ‘conversion’ meta-attributes.

Fig. 4. Definition of metamodel elements
Figure 5 shows the source template of our transformation model. As we can see, the transformation is performed, whenever a generalization relationship between two classes is found. Moreover, these two classes should be stereotyped as «userInput». The transformation is independent of the package in which the template elements are found (packageName = ?). Default name conversion for both of the classes is to add a prefix, and for their members - to copy them into the target elements.

Source model fragments that match template shown on Figure 5 will get transformed into model fragments matching template on Figure 6. This target template contains four TransformationElements representing four classes in the target model. After transformation, these classes will be related with a generalization, two aggregations and two associations. Their stereotypes will be set according to the template - to «dialogData», «XMLPacket» and «dialogWindow». These classes will be placed in appropriate packages (User Interface, Application Logic and Interfaces). It can be noted that name and member conversion is not defined, which suggests that the transformation is uni-directional (only from source to target).

The transformation model is completed with transformation rules, shown on Figure 7. These rules determine which source elements will get converted into
which target elements. The arrows show direction of this conversion (from source to target). Some TransformationLinks are adorned with constraints. These constraints override standard conversion rules defined in appropriate source TransformElements. As we can see, the UIDialog target elements will have names converted from the Specific source elements only (with a prefix, according to Figure 5). No members will be copied into UIDialog elements. On the other hand, the Data Transfer elements will have member lists being a concatenation of members from General and Specific elements.

The above described transformation model is the basis for performing the actual transformation on UML models. An example of such a model is shown on Figure 8. This model contains three classes stereotyped exactly as specified in the source transformation template. In this model we have two fragments that match correctly the template from Figure 5. It can be noted that these two fragments have a common General class - the UserData class.

After performing a transformation, the source model gets transformed into model shown on Figure 9. This diagram contains two fragments that are derived.
from the template on Figure 6. These two fragments are again joined by one of the classes (the CUserData class). The joining class is transformed from the joining class in the source model. It can be noted, that the class name has been extended with a prefix (letter 'C'). Both attributes of UserData have been copied into CUserData. Attributes from UserData have also been copied into XAdministratorData and XSSupervisorData. These two classes have also attributes copied from two other classes from the source model. This is consistent with the transformation rules shown on Figure 7.

The above example shows a very simple transformation of an analytical model into a design model. Three analytical classes are translated into seven design classes placed in different architectural layers (user interface, application logic, data transfer). This gets reflected in placing the resulting classes in three different packages that are shown as namespace prefixes in the class names.

5 Model management with a transformation tool

It is obvious that VTL specifications are meant for automatic transformations. Only an appropriate tool can effectively perform more complex transformations relieving developers from tedious copying and mapping classes between analytical and design packages. Figure 10 shows a typical task for a developer: take a model placed in a package on the analytical level and create a model placed in several packages on the design level. This second model could then be generated directly into code.
Fig. 10. Transformation in a CASE tool

This task can be partially automated if we can define standard transformation rules. With such rules present, a tool can apply these rules and perform appropriate transformation. Of course, the developer still would need to maintain the target model and add appropriate new attributes or operations to generated classes.

Figure 11 shows an application of a tool developed as a plug-in for a standard UML CASE tool. The plug-in allows to prepare a model of transformation written in VTL. Being an extension to UML, this model is stored inside the standard repository of the UML CASE tool. After defining all the source and target templates, and transformation rules, developer can invoke the transformation procedure. This procedure is quite simple, as it involves only choosing the source and target model packages. Additionally, the developer has to assign prefixes and postfixes to appropriate element transformations. After this, the plug-in tool takes the source model and matches it with source templates according to the transformation model. For each of the fragments in the source model that matches a template, appropriate transformation into the target model is performed.

6 Conclusions and directions for future work

Solving the problem of getting efficiently from user requirements to code seems to be crucial for the present software developers. Experience from various projects and case studies shows that this path can be significantly improved with the use

---

1 The tool used here is Enterprise Architect from SparxSystems which offers extensive and comprehensive programmer's interface.
of visual modeling techniques. With the proposed visual transformation language we receive another link in this modeling process. The language offers capability to define standard paths from the initial models that define the user needs, through architectural frameworks up to design solutions that can lead directly to code. It is very important that the language uses a very similar modeling notation as the models that are subject to transformations. This allows the developers to build visual templates for their transformations just as they would develop the actual models themselves.

Using MOF as a meta-language for VTL, and treating it as an extension of UML, gives us an advantage of the new language being compatible with the existing UML 2.0 CASE tools. It was relatively easy to use the existing repository structure of the chosen CASE tool (ie. the Enterprise Architect), to store the transformation definition together with the transformed models. This also allows for easy application of the same transformation to different models - simply by copying appropriate package with the transformation definition.

It has to be stressed that the proposed language does not solve all the problems associated with MDA/MDD [15] transformations. The target models still need a significant amount of human effort in order to accommodate them to appropriate technologies or architectural solutions. The future development of the VTL language, together with the transformation tool should go in the direc-
tion of extending the transformation rules into more detailed areas of particular UML models. With more detailed definition capabilities, most of the development effort could go into defining transformations. These could be defined only once for all applications in the given problem domain or using the same technology. With these transformations in place, the majority of development would be concentrated on creating requirements consistent with the real user needs. The rest could be performed almost automatically - with a little intervention of the developers. However, this is still to come...

References