Generic XMI-Based UML Model Transformations

Jernej Kovse, Theo Härder

Department of Computer Science
University of Kaiserslautern
P.O. Box 3049, D-67653 Kaiserslautern, Germany
{kovse, haerder}@informatik.uni-kl.de

Abstract. XML-based Metadata Interchange (XMI) is an interchange format for metadata defined in terms of the MOF standard. In addition to supporting the exchange of complete models, XMI supports the exchange of models in differential form. Our paper builds on this feature to examine the possibility of XMI-based generic transformations of UML models. A generic transformation can be configured to generate (via XSLT) a specialized transformation that will be used to transform a UML model. The approach promotes model reuse, speeds up the modeling process and can be used to assure that only predefined semantics (as specialized by an agent) is included in the transformed model.

1 Motivation

The XML-based Metadata Interchange (XMI) [11] is an interchange format for metadata that is defined in terms of the Meta Object Facility (MOF) [8] standard. Since the adopted UML specification [10] defines the UML meta-model as a MOF meta-model, XMI can be used as a model interchange format for UML. This allows UML modeling tools or repositories from different vendors to use XMI to exchange UML models. In addition to supporting the export or import of complete models or model fragments, XMI allows the exchange of models in differential form, i.e. in case a modeling tool is used to extend a UML model m_1 with new model constructs to produce model m_2, XMI supports the exchange of differences between the two models. An important consequence of this feature is that XMI can be used to describe model transformations.

This paper[1] builds on this feature of XMI to examine an infrastructure needed for semi-automatic transformations of UML models in a UML repository. A general usage scenario for such transformations looks like this: A UML model m_i is given and stored in a UML repository. A human or a software agent wants to transform m_i, i.e. add, remove, or modify model elements to obtain a model m_{i+1} which expresses the new (extended, deprived or modified) semantics. However, suppose completely manual model transformation by an agent is unacceptable - we would like to assure that the semantics contained in m_{i+1} is understood by a UML model-driven compiler (e.g. a model-based software generator [2]) or interpreter (e.g. a workflow engine based on

[1] The research is part of Sonderforschungsbereich (SFB) 501, funded by the Deutsche Forschungsgemeinschaft (DFG).

UML activity graphs [4]). As a solution to this problem, we half-fabricate a model part (the difference between m_{i+1} and m_i) and represent it as a generic transformation. Generic transformations are stored in a database that allows the agent to query them and select the one matching its design requirements. By configuring a generic transformation, the agent produces an XMI document describing the transformation that has to be applied to obtain the model m_{i+1}. The process of selecting different generic transformations, configuring them and transforming the model can be iterated to meet different requirements that have to be present in the final model. Generic transformations promote model reuse, speed up the modeling process and assure that only predefined semantics (as specialized by the agent) is integrated in the final UML model.

Section 2 of this paper focuses on the required infrastructure supporting the application of XMI-based generic transformations in detail. Section 3 gives a brief overview of related work. In Section 4, we make a conclusion and present some ideas for the future work related to the approach.

2 An Infrastructure for Generic XMI-Based Transformations

2.1 XMI-Supported Model Transformations

XMI defines four elements (we refer to them as differential elements) used to support differential description of UML models: `XMI.difference` is the parent element used to describe differences from the base (initial) model; it may contain zero or more differences, expressed through the elements `XMI.difference`, `XMI.delete`, `XMI.add` and `XMI.replace`. `XMI.delete` represents a deletion from the base model, `XMI.add` represents an addition to the base model and `XMI.replace` represents a replacement of a model construct with another model construct in a base model.

2.2 Generic Transformations

Suppose a set of transformations $\{T_k;\ k\in[1,n]\}$, which may all be derived (specialized) from a common transformation blueprint. We call such a blueprint a generic transformation `T`. `T` gives a generic view on a modular part of a UML model whereas each T_k represents a full specialization of this part and allows its integration with an existing base model m_i. T_k can be generated from `T` by configuring `T`'s parameter values. To allow a direct application of T_k to m_i, T_k is expressed as a series of XMI's differential elements. The configuration parameters (specified by an agent) define how a transformation generator generates T_k from `T` using the following three mechanisms (Fig. 1 illustrates an example of two specialized transformations generated from the same generic transformation where `Z` is the class added in each transformation):

- *Static parameterization*: Parameters of `T` are used to generate concrete properties of differential elements used in T_k, e.g. class names, association names, role names, multiplicities, visibility kinds, definitions of OCL constraints, etc. The designer of

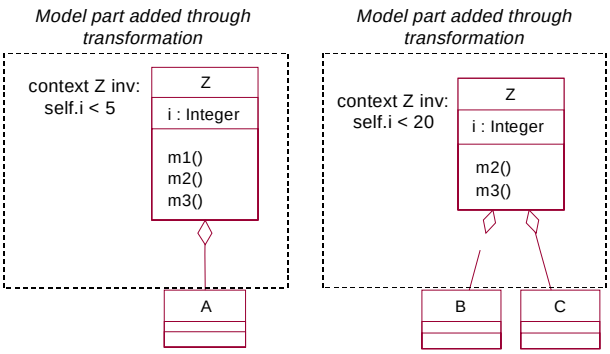

Fig. 1. A pair of specialized transformations

`T` defines places (templates) in `T` that are used as placeholders for parameter values. For example, an invariant limits the values of `i` to two different ranges in Figs. 1a and 1b.

- *Iteration*: Parameters of `T` allow the agent to influence how many times a segment of `T` will be used in T_k. For example, in Fig. 1a, `Z` acts as a container for `A` whereas in Fig. 1b, it acts as a container for classes `B` and `C` meaning that two aggregation associations had to be created.
- *Conditional include*: Parameters of `T` allow the agent to decide whether a segment of `T` will be used in T_k. For example, in Fig. 1b, we have omitted the method `m1` from class `Z`.

It is the responsibility of the designer of `T` to carefully consider how to combine the above mechanisms so that T_k could be generated and applied in a meaningful (consistent) form. To achieve this, the designer always has to specify parameter guidelines (described in Section 2.3) used to limit the set of specialized transformations that can be derived from the same generic transformation `T`.

T_k is generated as an XMI document (referred to as `Tk.xml`) using two sources: A definition of `T` (provided by the designer of `T`) and a list of parameter values for `T` (provided by the agent). We suggest that both sources should be provided as XML documents, referred to as `T.xml` and `parameters.xml`, respectively. In this case, XSLT (Extensible Stylesheet Language Transformations) [14] can be used by the transformation generator to process `T.xml` and `parameters.xml` to generate `Tk.xml`. XSLT is a language for transforming XML documents (source trees) into other XML documents (result trees). An XSLT transformation, expressed as an XSLT stylesheet (referred to as `generate.xslt`), specifies rules used by the transformation generator to generate `Tk.xml` from `T.xml` and `parameters.xml`. `generate.xslt` is used to support the identified three mechanisms in XSLT.

- Static parameterization is supported using XSLT template rules (element `xsl:template`) [14].
- Iteration (see example in Fig. 2) can be supported either by XSLT template rules or XSLT repetition (element `xsl:for-each`) [14].
- Conditional include is supported using XSLT conditional processing (`xsl:if` and `xsl:choose`) [14].

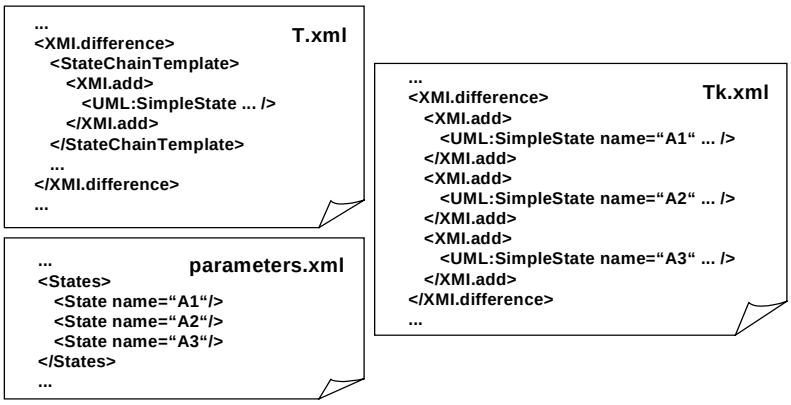

Fig. 2. Iteration (supported via XSLT template rules or repetition)

2.3 Constraints

We cannot always rely on the agent to make the appropriate judgment on whether T_k can be consistently applied to m_i. For this reason, the designer of `T` defines two sets of constraints for each generic transformation: *parameter guidelines* define valid configurations of parameter values in `parameters.xml` and are thereby used to limit the set of transformations that can be derived from `T`. *Transformation preconditions* define constraints that apply to m_i and assure that the constructs it contains allow a consistent application of T_k. Since they apply to a UML model, transformation preconditions are expressed as OCL constraints [10].

2.4 Applying Transformations

As `Tk.xml` is generated, the model transformer attempts to apply it to the model m_i. The infrastructure includes a UML repository [6] that stores UML models. A repository [1] offers several benefits for the process of applying transformations. Since it supports version control, the models obtained by applying consecutive transformations can be represented as versions. This allows an agent to revert to a previous version of the model even after a transformation has committed. Since a repository supports configuration control, it allows multiple model segments that have been transformed independently to be combined into configurations. Notification services allow UML-model driven compilers or interpreters (see Section 1) to be notified whenever a model change (that they have as listeners registered to) occurs.

The UML repository offers the access to stored UML models via a programming model [1], which is a mapping of the UML metamodel [10] to enterprise components. The components expose the core Create-Read-Update-Delete (CRUD) methods for the UML model elements. A programming model based on enterprise components delivers persistence, scalability (to leverage the performance of complex model transformations) and programming level transactions. The model transformer is implemented as a session component that performs a transformation T_k in the following

order: (i) it initiates a transformation transaction, (ii) checks m_i for transformation preconditions, (iii) parses `Tk.xml` to translate the occurences of differential XMI elements to the invocations of CRUD methods of components of the repository's programming model, (iv) commits the transformation transaction.

3 Related Work

St-Denis et al. [12] compare various model interchange formats, e.g. RSF, XIF, XMI, and discuss the implementation details of an XMI-based model interchange engine. They identify the XMI's support for differential model exchange as vital for the scalability, which is one of the requirements they use to assess the formats.

Keienburg and Rausch [5] present an infrastructure for model evolution, schema migration and data instance migration, which is based on UML models. Successive differences on the evolution path are represented using the XMI's differential elements.

Yoda [15] presents an approach to developing applications using parameterized frameworks. The approach applies to the OMG's Model Driven Architecture (MDA) [9]. He recognizes model transformations as a way to customize predefined and parameterized frameworks. The parameterized UML diagrams he presents can be compared to the mechanism of static parameterization we identified in Section 2.2. Yoda classifies the parameterized frameworks as attribute- or operation-centric.

Demuth et al. [3] outline XMI-based scenarios in the forward and reverse engineering of different applications. As an example, they show how XSLT can be used to generate a SQL schema from a UML model.

Schema transformations are extensively discussed by McBrien and Poulovassilis [7]. In case an evolving UML model has to be consecutively mapped to an (object-)relational database schema, issues related to schema evolution, discussed by Türker [13], become highly relevant to the proposed approach.

4 Conclusion and Future Work

This paper presented our work on generic XMI-based transformations of UML models. The proposed infrastructure allows agents that want to transform UML models to select a predefined generic XMI-based transformation and configure it via parameter values. A specialized XMI transformation is generated by using XSLT as a mechanism for transforming XML documents. The generated transformation is applied in a UML repository that allows versioning of successively transformed models and formation of model configurations from independently transformed model parts.

The target applications of the proposed approach are UML model-driven compilers and interpreters that can understand the UML-specified semantics only in predefined ways. Thus, agents using them can apply generic transformations to build UML models that conform to their requirements. The approach also promotes model reuse, speeds up the modeling process and assures that only predefined semantics is included in the final models.

As future work, we intend to examine whether automatic identification of generic XMI-based transformations is feasible: Given a base UML model m_i and a set of transformed UML models, $\{n_j, j \in [1,l]\}$, what are the possibilities to automatically extract a generic transformation that would allow any model from the set to be generated from m_i.

Second, we are interested in the extension of the proposed infrastructure that would support model-based generative software development. This paper has handled semi-automatic transformations of UML models. In our future work, we will try to identify how the current concepts of generative programming, e.g. feature modeling, template metaprogramming, aspect-oriented programming [2] fit into the proposed infrastructure that would support semi-automatized implementation of software parts from the UML models that have been successively enhanced using generic XMI-based transformations.

References

[1] Bernstein, P.A.: Repositories and Object Oriented Databases, Proc. Conf. BTW'97, Ulm, March 1997, Springer-Verlag, pp. 34-46.

[2] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000.

[3] Demuth, B., Hussmann, H., Obermaier, S.: Experiments with XMI-based Transformations of Software Models, in: Online Proc. WTUML: Workshop on Transformations in UML (ETAPS 2001 Satellite Event), Genova, Apr. 2001, http://ase.arc.nasa.gov/wtuml01/

[4] Dumas, M., ter Hofstede, A.H.M.: UML Activity Diagrams as a Workflow Specification Language, in: Proc. Int. Conf. UML 2001, Toronto, Oct. 2001, Springer-Verlag, pp. 76-90.

[5] Keienburg, F., Rausch, A.: Using XML/XMI for Tool Supported Evolution of UML Models, in: Proc. Int. Conf. HICSS'01, Maui, Jan. 2001, IEEE, 2001.

[6] Mahnke, W., Ritter, N., Steiert, H.-P.: Towards Generating Object-Relational Software Engineering Repositories, in: Proc. BTW'99, Freiburg, March 1999, Springer-Verlag, pp. 251-270.

[7] McBrien, P., Poulovassilis, A. : A Formal Framework for ER Schema Transformation, in: Proc. ER'1997, Los Angeles, Nov. 1997, Springer-Verlag, pp. 408-421.

[8] OMG: Meta Object Facility Specification, version 1.3.1, OMG document 01-11-02.

[9] OMG: Model Driven Architecture - A Technical Perspective (Draft), OMG document 01-07-01.

[10] OMG: Unified Modeling Language Specification, version 1.4, OMG document 01-09-67.

[11] OMG: XML Metadata Interchange Specification, version 1.2, OMG document 02-01-01.

[12] Saint-Denis, G., Schauer, R., Keller, R.K.: Selecting a Model Interchange Format. The SPOOL Case Study, in: Proc. Int. Conf. HICSS'00, Maui, Jan. 2000, IEEE, 2000.

[13] Türker, C.: Schema Evolution in SQL-99 and Commercial (Object-)Relational DBMS, in: Proc. 9th Int. Workshop on Foundations of Models and Languages for Data and Objects (FoMLaDO 2000), Dagstuhl, Sep. 2000, Springer-Verlag, pp. 1-32.

[14] W3C, XSL Transformations (XSLT), version 2.0 (Working draft), http://www.w3.org/TR/xslt20/

[15] Yoda, T.: Creating Applications Using Parameterized Frameworks: Quickly developed and highly customized, presentation at OMG's 2nd Workshop: UML for Enterprise Applications: Model Driven Solutions for the Enterprise, Burlingame, Dec. 2001.