Generic and Meta-Transformations for Model Transformation Engineering*

Dániel Varró and András Pataricza

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Budapest, Magyar tudósok körútja 2.
{varro,pataric}@mit.bme.hu

Abstract. The Model Driven Architecture necessitates not only the application of software engineering disciplines to the specification of modeling languages (*language-ware*) but also to design inter and intra-language model transformations (*transformation-ware*). Although many model transformation approaches exist, their focus is almost exclusively put on functional correctness and intuitive description language while the importance of engineering issues such as reusability, maintainability, performance or compactness are neglected. To tackle these problems following the MDA philosophy, we argue in the paper that model transformations should also be regarded as models (i.e., as data). More specifically, we demonstrate (i) how *generic transformations* can provide a very compact description of certain transformation problems and (ii) how *meta-transformations* can be designed that yield efficient transformations as their output model.
Keywords: model transformation, metamodeling, meta-transformation, generic transformation.

1 Towards Model Transformation Engineering in MDA

MDA and language engineering. Recently, the Model Driven Architecture (MDA) of the Object Management Group (OMG) has become a dominant trend in software engineering. The main idea of the MDA framework is the use of models during the entire system design cycle. At first, the central business logic functionality on the target system is captured by the so-called platform-independent model (PIM). Information on the target software platform is added in a later phase when mapping PIMs into platform-specific models (PSMs). Finally, the entire source code of the target application can be generated automatically.

A key factor in the success of the MDA is thus the development of industrial-strength models in various modeling languages. Several metamodeling approaches [3, 6, 10, 24] have been flourishing to provide solid foundations for language engineering (or language-ware) to allow systems engineers to design a language for their own domain. As being the standard and visual object-oriented modeling language, UML obviously plays a key role also in language design.

* This work was partially supported by the Hungarian National Scientific Foundation Grant (OTKA 038027).

Transformation engineering in MDA. Many methodologists and computer scientists have recently pointed out that the role of model transformations between modeling languages within MDA is as critical as the role of modeling languages themselves [4]. MDA requires various kinds of transformations including inter-model (for instance, from PIMs to PSMs) and intra-model transformations (such as transformations within PIMs), or model to code mappings. As these transformations will be mainly developed by software engineers, precise yet intuitive notations are required for model transformation languages. QVT [17], a recent initiative of the OMG, aims exactly at developing a standard for capturing Queries, Views and Transformations in MDA.

Many model transformation approaches exist (including the official proposals submitted to the QVT RFP and overviewed in [12]). An "ultimate debate" that separates these approaches is about the way of specifying transformations: declarative approaches (like [2, 13, 16]) define a relation between elements of the source and target modeling language while operational approaches (such as [14, 20, 23, 25, 26]) define rules to describe what steps are required to derive the target model from a given source model. Mixed approaches (see [21]) typically use declarative relations first which are manually refined into operational rules later on. In general, declarative approaches tend to be more intuitive for software engineers (i.e., it is easier to write and understand declarative transformations) while it is easier to automate operational approaches (i.e., it is easier to develop tools that execute operational transformations).

Problem statement. Due to the central role of model transformations in the MDA framework, transformations should be developed within MDA (thus, for instance, by distinguishing between platform-independent and platform-specific transformations as proposed in [4]). This also necessitates adapting the well-known principles of software engineering for transformation engineering (or *transformation-ware*, shortly, *transware*) to support the entire design cycle of model transformations. Despite the wide range of existing model transformation approaches, their focus is almost exclusively put on developing functionally correct model transformations using an intuitive notation while neglecting the importance of traditional software engineering issues such as reusability, maintainability, performance or compactness.

Objectives. To tackle (many of) these problems in a convenient way complying with the MDA philosophy, we argue in the paper that model transformations should also be regarded as models (or in other terms, as data). More specifically, we demonstrate (i) how *generic* (or higher-order) *transformations* can provide a very compact description of certain transformation problems in MDA (Sec. 3) and (ii) how *meta-transformations* (Sec. 4) can be designed that yield efficient transformations as their output model using VPM (Visual Precise Metamodeling), a multilevel and dynamic (rule-based) metamodeling framework. The concepts are demonstrated on a well-known (meta-)transformation problem of the MDA framework. Finally, we also discuss some tooling aspects for supporting our concepts (Sec. 5).

Proc. UML 2004: International Conference on the Unified Modeling Language
Lisbon, Portugal, October 10–15, 2004.

2 Motivating example: Generating XMI models

In order to motivate the need for generic and meta-transformations in MDA, we selected a well-known transformation problem, namely, generating standard XMI documents from MOF-based models. The relevance of this problem is also demonstrated by the fact that an official QVT submission [21] discusses this transformation for handling one specific modeling language, namely, UML.

However, it is worth pointing out that the XMI standard [18] is not related only to UML. In fact, it is a meta-standard in the sense that it specifies a corresponding XML format for an arbitrary modeling language defined by its MOF metamodel [19]. Although XMI is intended to serve as a common interchange format for the MOF-based models of UML tools, the XMI formats of these tools are essentially different from each other (despite the unique and standard UML metamodel). The lack of proper tool support for XMI export and import in off-the-shelf UML tools is a main motivation for discussing this transformation in the current paper. More specifically, we argue that capturing the XMI transformation in its generality is much easier than specifying it just for a single modeling language such as UML.

For an overview, a simplified variant of the MOF metamodel (i.e., the meta-metamodel in the MOF framework), and an XML metamodel are depicted in Fig. 1. Furthermore, an extract of a metamodel for the language of graphs (which is an instance of the MOF metamodel) and a sample XMI document are depicted in Fig. 2.

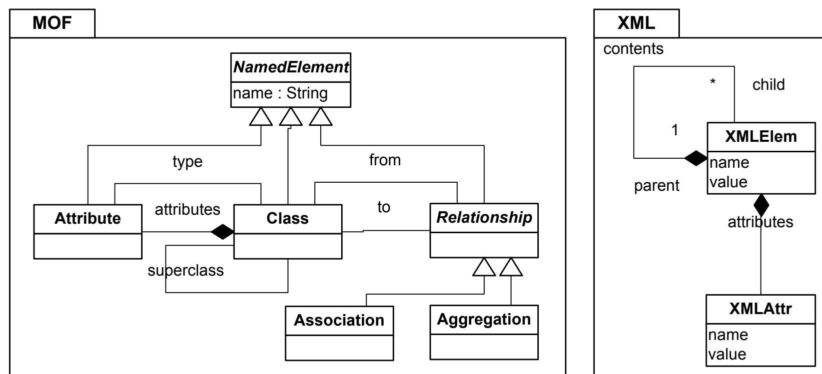

Fig. 1. The metamodels of MOF and XML

The main concepts of the MOF-model-to-XMI transformation are described informally below. For space limitations, we disregard the handling of inheritance, the navigability of association ends, and the sequencing of XML elements.

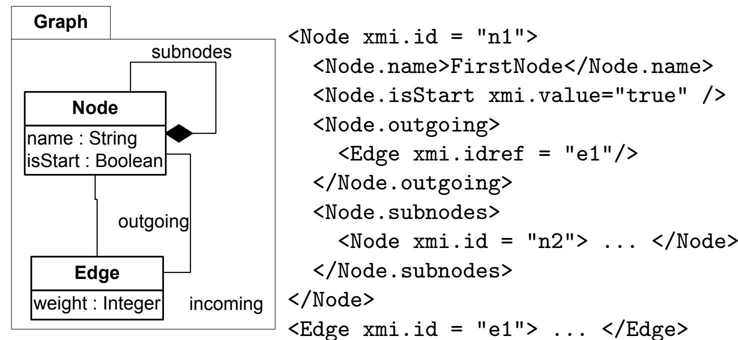

Fig. 2. A sample modeling language and XMI document

- The MOF metamodel of a language (e.g., the language of graphs in Fig. 2) defines the names and the structure of XML elements and attributes for the corresponding XMI format of the modeling language.
- Each instance of a MOF class *cls* (i.e., each object in traditional UML terminology) is transformed into a separate XML element (i) with a name corresponding to the name of the class (*cls*) and (ii) with a special attribute called *xmi.id* for storing the identifier of the instance. See the encoding of node *n1* in Fig. 2 for demonstration.
- Each instance of a MOF attribute *att* in a MOF class *cls* is mapped into an XML element with a name *cls.att* and placed inside the start and end tags of its owner element. If the type of an attribute is an enumeration type, then we use the special XML attribute *xmi.value* to store the value of the attribute instance (see attribute *isStart*). Otherwise, the value is stored within the start and end tags of the corresponding XML element as a string (as done in the case of attribute *name*).
- Each instance of a MOF aggregation is transformed into an XML element that encapsulates all the objects inside the container object (we assume that each object can be contained by exactly one other object to satisfy the tree criteria for XML models). XML elements of contained objects are placed inside this intermediate XML element representing the aggregation itself (see the aggregation *subnodes* in the example).
- Each instance of a MOF association is mapped into (i) an XML element corresponding to the association itself (such as *outgoing*) and (ii) an XML element corresponding to the target object at the other end of the association instance which contains a reference to the main XML element of the object appearing elsewhere in the XML document (see edge *e1*). This way, the complex graph structure of a MOF-based model can be encoded into the strict tree structure of an XML document.

3 Generic Transformations

Generic (or higher-order) transformations contain transformation rules where the types of certain objects are variables. An analogy can be made between general transformations and generic (or template) classes used in various object-oriented languages. However, while the parameters of generic classes are bound at design time, type variables in generic transformation rules are only substituted at transformation-time (run-time).

The advantages and drawbacks of higher-order transformations also show certain similarities with traditional logic frameworks. Higher-order logic is a very powerful description mechanism but it raises decidability (and performance) problems concerning automated reasoning when compared with traditional first-order logic. Still, several powerful (higher-order) theorem provers have been applied for a large scale of practical verification problems.

Analogously, generic transformation rules offer a very high level of generality and compactness when compared to other (first-order) model transformation frameworks (especially, for higher-level transformations). A single generic rule can handle several situations where essentially the same rule pattern should be applied on objects of different types. On the other hand, the foundations of their type system (metamodeling framework) require some precautions to avoid certain well-known problems (see [3]). Furthermore, degradation in performance has been experienced in rewriting logic systems like Maude [7] when working with the meta-representations of large models.

To overcome these problems, we will build on VPM [24], which is a dynamic metamodeling framework with fluid meta-levels. Moreover, generic transformation rules will be turned into traditional first-order ones by meta-transformations later on in Sec. 4 to tackle performance problems.

3.1 The metamodeling and transformation framework

In the paper, we use the VPM metamodeling framework [24] to provide the theoretical foundations for our concepts. VPM is a multilevel framework in the sense that it introduces an explicit representation of both instance-of and inheritance relations, which is extended to handle associations, models and metamodels (in addition to classes) but the notions of metalevels are fluid. Furthermore, VPM is a dynamic framework, where inter and intra model transformations can be described by a visual, rule and pattern-based formalism provided by the paradigm of graph transformation [9].

VPM provides the following important features for supporting generic and meta-transformations.

– The instance-of relations between model elements are stored and manipulated as any other model elements (provided that the axioms of [24] are fulfilled). As a result, classes and objects can be represented uniformly as entities (nodes) with an explicit instance-of relation between them in order to avoid the "redefinition of concepts" problem [3].

- Transformation rules are also stored as ordinary models (or, in other terms, as data), therefore, we can easily develop transformation rules that process or generate transformation rules.

A transformation rule (see, e.g., *Obj2ElemR* in Fig. 3) is represented as a package having a source and a target pattern inside. Each entity in the pattern is either a constant or a variable (constants start with upper case initials or printed within in quotation marks). Correspondence between source and target elements are denoted by pentagons such as *Obj2Elem(o,e)* which is only syntactic sugar for correspondence objects of a specific class defined by the metamodel of the mapping (see [13] for the metamodeling of mappings).

A transformation rule can be applied on a given model by a transformation engine which first tries to match all the elements in the rule (patterns) that are not marked by {*new*} labels to model elements. A constant entity in the rule should be matched to the entity with the same identifier in the model while a variable entity can be matched to any type and edge-conforming entity of the model. Then, the images of all elements marked by {*delete*} labels in the rule are removed from the model (potentially including the removal of certain dangling edges). Finally, new nodes and edges are created that correspond to elements marked by {*new*} labels in the rule.

Instance-of relations are depicted either *explicitly* by dashed edges between two entities (an object and its class) and/or *implicitly* by using entity names of type *obj:class* to denote the fact that entity *obj* is an instance of entity *class*. However, it is crucial to point out that in generic transformation rules, both entities *obj* and *class* can be either constants (i.e., concrete objects or classes) or variables independently. For instance, a pattern where both *obj* and *class* are variables can be matched to all instances of all classes. If only *class* is a variable then pattern matching finds the type (or all types) of an object. Finally, if *class* is constant and *obj* is variable, then we have traditional first-class patterns. Note, however, that the type hierarchy of each rule should be upper closed, i.e., each entity should have (directly or indirectly) a constant entity as a type.

3.2 Details of the MOF-XMI transformation

The rules of (a fragment of) the MOF-model-to-XMI transformation are now presented in Fig. 3 and 4 and explained below. The entire case study (that handles inheritance, the sequencing of XML elements, etc.) consists of less than 20 rules.

- *Obj2ElemR* (in Fig. 3). First each object *o* in our MOF-based model that is an instance of a MOF Class *x* in the metamodel is transformed into a corresponding XML element *e* with an XML attribute *a* (called *xmi.id*). In fact, *x* is, in turn, an instance of *Class* in the MOF metamodel as denoted by *x:Class*. The name of *e* equals to the name of class *x* while the identifier of *o* is mapped into the value of attribute (instance) *a*. The XML constructs *e* and *o* and the *Obj2Elem(o,e)* correspondence is generated as a result of the rule application.

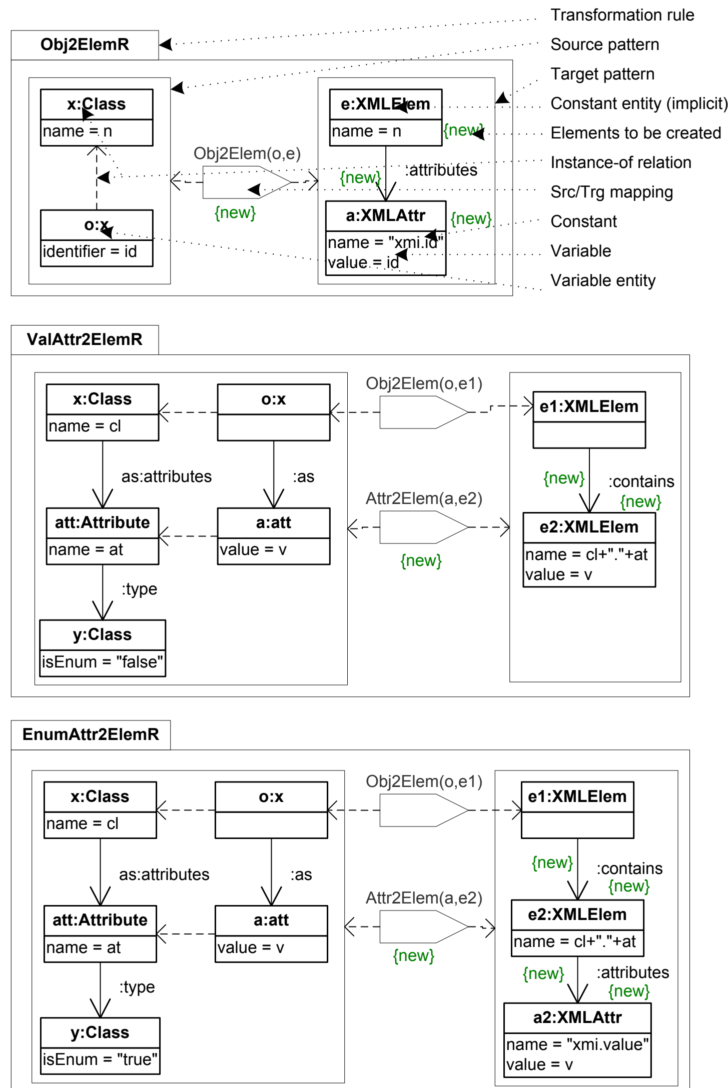

Fig. 3. Transforming objects and attributes

– *ValAttr2ElemR* (in Fig. 3). The handling of MOF attribute instances are dependent on the metamodel. If the type of an attribute *att* (in a class *x*) is not an enumeration type (i.e., *isEnum* = *false*) then the value of an attribute instance *a* is stored as a string between the start and end tag of an XML element *e2* and it is related to attribute instance *a* (by the *Attr2Elem(a,e2)* relation). Furthermore, we add *e2* to the content of the XML element *e1*

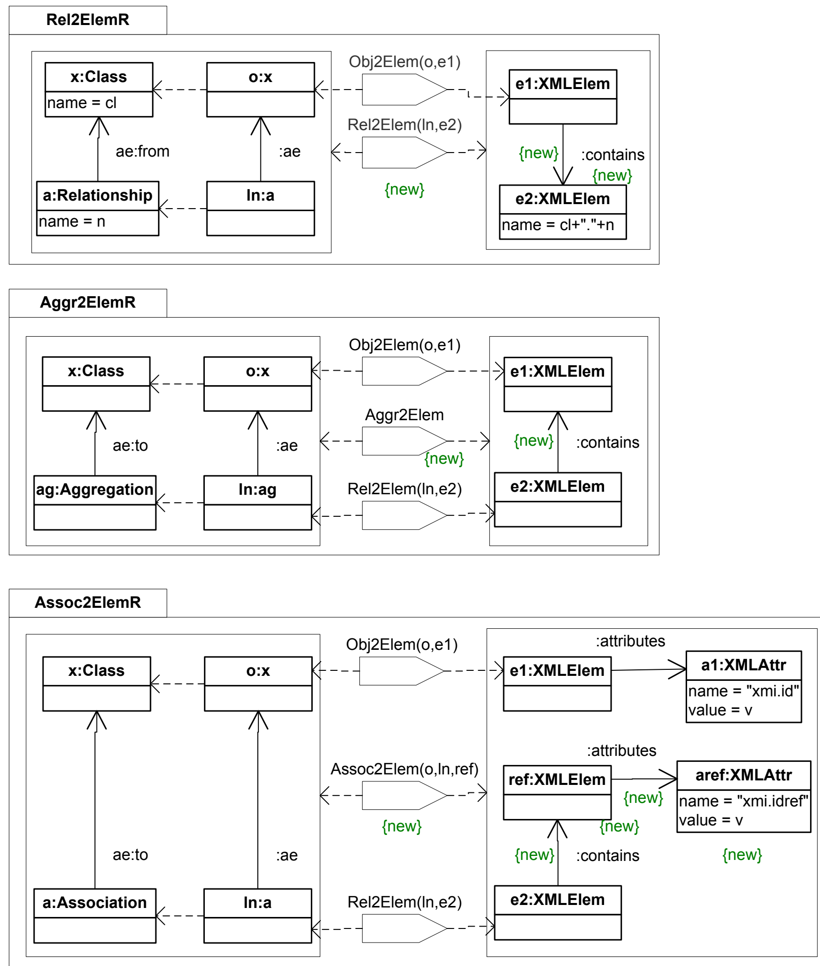

Fig. 4. Transforming associations and aggregations

which corresponds to the container object *o* of the attribute instance *a* (as denoted by checking the existence of an *Obj2ElemR(o,e1)* relation).

– *EnumAttr2ElemR* (in Fig. 3). If the type of an attribute *att* (in a class *x*) is an enumeration type (i.e., *isEnum* = *true*) then the value of an attribute instance *a* is stored in an XML attribute (*a2*) called *xmi.value* that belongs to XML element *e2*. Note that in both cases, the names of the new XML elements are created by concatenating the name of the class and the attribute (see *Node.isStart* and *Node.name* in Fig. 2).
– *Rel2ElemR* (in Fig. 4). For the sake of simplicity, both associations and aggregations are handled as directed relationships (with a source/from and a

target/to class; see the MOF metamodel in Fig. 1). Rule *Rel2ElemR* handles the source part of relationships. For a relationship instance *ln* it creates a new XML element *e2* and links it to an XML element *e1* that corresponds to the source object *o* of the relationship instance *ln*. The names of new XML elements are derived again as the concatenation of the names of the source class and the relationship.
- *Aggr2ElemR* (in Fig. 4). This rule simply places XML elements *e2* corresponding to the target object *o* of an aggregation instance *ln* into the (contents of) XML element *e1* which is related to the aggregation instance *ln* itself.
- *Assoc2ElemR* (in Fig. 4). Finally, in case of associations (references), we look up the corresponding XML element *e1* of the target end object *o*, and generate a new XML element *ref* referring to *e1* by the value of its *xmi.idref* attribute (see the main and the reference Edge element in Fig. 2).

As a conclusion, generic (graph) transformation rules provide an extremely compact and powerful platform-independent specification mechanism for model transformation problems in the MDA framework, especially, when more than two metalevels are involved in the transformation itself (as in case of the MOF-to-XMI transformation).

However, an increase in generality (naturally) causes a decrease in performance as experienced also in other meta-theoretical frameworks such as in [7]. Even if a clever pattern matching strategy is implemented (e.g., matching type variable nodes before instance variable nodes), higher-order transformations are necessarily slower than traditional first-order ones. To tackle performance (and maintainability) problems, we now introduce meta-transformations that will derive efficient, first-order from generic (higher-order) transformations.

4 Meta-Transformations

Meta-transformations are transformations that operate on other transformations as their input or output. A prerequisite for meta-transformations is thus to store transformation rules also as ordinary models within the MDA framework. This idea is in direct analogy with the Neumann architecture of computers where programs are stored as data and only interpreted differently by the target machine.

Increasing performance of generic rules. A main goal of using meta-transformations is to increase the performance of generic transformations. Essentially, the pattern matching of a generic rule is split into two separate phases by creating rule-writing rules. Type variables are substituted in all possible ways, and each substitution yields a first-order transformation rule (which now only contains instance variables). Rule-writing rules are again higher-level, but (i) they are executed at design time, and (ii) they operate on the metamodels, which are several orders of magnitude smaller than the (system) models themselves, thus performance is not critical. Then, the main (model-level) transformation itself is carried out at transformation-time by efficient first-order rules.

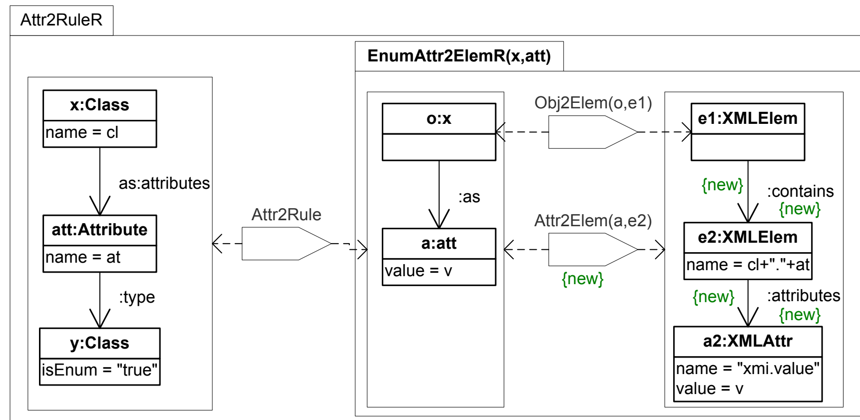

Fig. 5. A meta transformation rule for generating rules

A sample meta transformation rule that tailors the generic rule *EnumAttr2ElemR* to first-order rules is depicted in Fig. 5. When applied to the metamodel of graphs in Fig. 2, variables *x*, and *att* are substituted, therefore, the type variables in the target rule model *EnumAttr2ElemR(x,att)* are bound, for instance, to class *Edge* and attribute *weight*. In this respect, the generated rule (i.e., the target model of the meta-transformation) is now a traditional first-order rule (see Fig. 6) that can be used in any graph transformation engine. However, this rule was generated automatically which is much less error-prone than a copy-paste method with manual substitution of type variables.

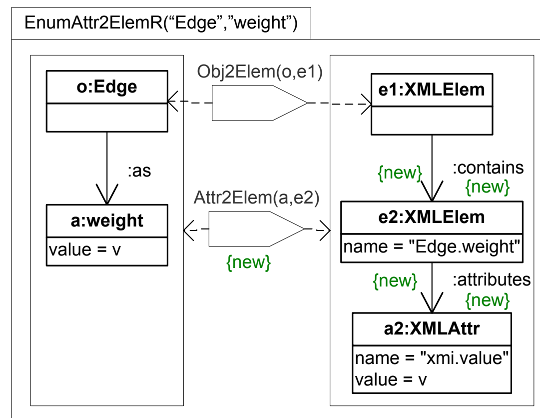

Fig. 6. A rule generated by applying the meta rule of Fig. 5 for the language of graphs

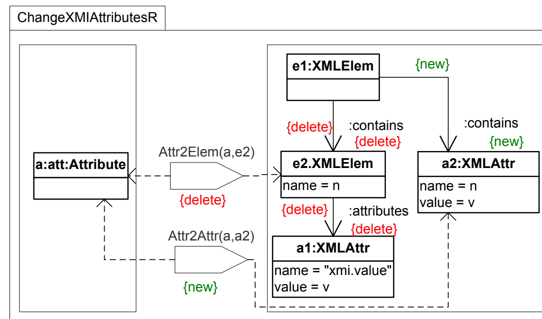

Fig. 7. Maintaining XMI models and rules

Increasing maintainability of transformations. An alternate main goal for the use of meta-transformations is concerned with the maintainability of transformation rules. As a motivation, it is not hard to imagine that we need to adapt our MOF-to-XMI transformation in practice to the XMI dialects of various tools (especially, as XMI versions 1.0 and 1.1 are very different).

For instance, a new XMI dialect no longer represents instances of MOF attributes as XML elements but as XML attributes (as done in XMI 1.1 to reduce the number of elements in the XML tree). Therefore, we created the rule of Fig. 7 to migrate the attribute changes of XMI 1.0 models into XMI 1.1 models. This rule also prescribes the deletion of certain model elements (such as the superfluous XML element *e2* for representing an attribute instance or its corresponding *xmi.value* attribute *a1*) in addition to creating a new XML attribute *a2* for storing the value of the attribute.

Now the rule of Fig. 7 can be applied directly on XMI 1.0 models to convert them into their XMI 1.1 equivalent. However, what is more interesting, we can use transformation rules (of Fig. 5-6) as source models of this maintenance rule so that the rules generating the XMI output are updated themselves. As a result, the same rule can be used for the maintenance of both transformation models and rules.

As a conclusion, meta-transformations are extremely powerful means (i) to increase the performance of generic rules by tailoring them into first-order rules of a specific domain, and (ii) to maintain or upgrade model transformations. These are achieved by treating transformation rules as ordinary models.

5 Tool support

We have recently made major upgrades and re-engineering of VIATRA [25], our general model transformation framework, in order to support the dynamic, multilevel metamodeling features of VPM [24], and the generic/meta transformational foundations presented in the current paper. The main intended usage of

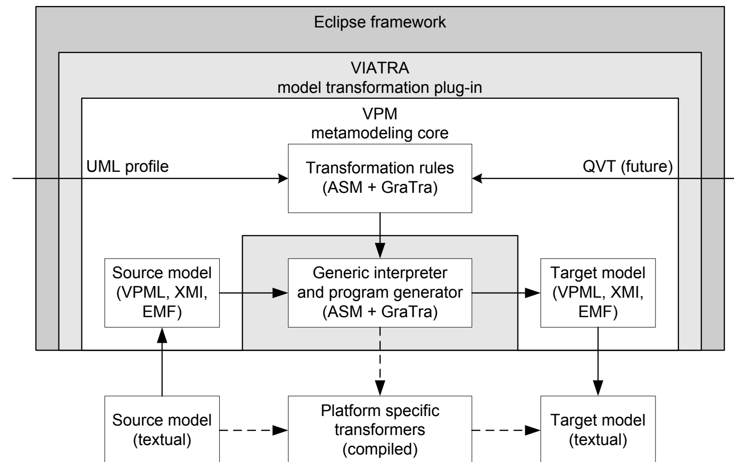

Fig. 8. The VIATRA model transformation architecture

our framework is dependability evaluation and optimization of business workflow models and UML models.

The VIATRA 2.0 framework (see Fig. 8) is intended to be used in two alternate ways: platform-independent and platform-specific way.

– *Platform-Independent Transformations.* A source user model (which is a structured textual representation such as an XMI description of a UML model exported from a CASE tools) is imported into the VPM model space. Then, platform-independent transformation specifications can be constructed by combining graph transformation [9] and abstract state machine [5] rules. These rules can be created within the framework or in a UML tool using a special profile (and, in the future, using the QVT standard). The rules are then executed on the source VPM model by the generic (higher-order) VIATRA rule interpreter in order to yield the target (VPM) model. Finally, the target model can be serialized into an appropriate textual representation specific to back-end tools. This way, the transformation is kept within a single transformation framework (i.e., VIATRA) in order to ease the testing, debugging and validation of model transformations without relying on the highly optimized target transformation technology.
– *Platform-Specific Transformations.* The VIATRA engine also enables the design of meta-transformations that take an already validated transformation specification as the input and yield a platform-specific transformer (e.g., a Java program, or XSLT script) as the output. In other terms, the functionality of a transformation program is *compiled* into a more efficient (but less general) target transformation technology. In this respect, the transformation from a specific source model to its target equivalent can be performed outside the VIATRA framework. This is especially important for integrating

complex transformations into off-the-shelf CASE tools which, normally, have their own, tool-dependent way for writing transformation add-ons.

A main technological change is that VIATRA 2.0 is now implemented as a plug-in for the Eclipse framework [1] (in contrast to the previous Prolog version).

6 Related work

The main point of the current paper is to draw attention to the fact that important MDA transformations can only be properly captured as a generic and/or meta-transformations. However, these issues are not addressed in the QVT RFP (thus not in any of the submissions), we can only find some weaker constructs. For instance, the use of multi-objects in patterns (e.g., in the QVT partners proposal [21]) still can only handle transformations on the same metalevel (while variables may appear on any metalevels in our approach). Using both classes and objects in transformation rules in a UML style is rather higher-level, but our approach can also handle an arbitrary number of metalevels, thus models can be taken from various technological spaces (with different meta-metamodels).

Existing graph transformation approaches and tools (e.g. AGG [11] or ATOM3 [8]) work with a fixed metamodel (type graph) for a given transformation disallowing higher-order variables. Only Progres [22] allows the use of type parameters in rules to support higher-order transformations, but meta-transformations are still not supported.

Probably, the closest theoretical correspondence is provided by (i) two-level graph grammars [15] introduced for the definition of context-sensitive language grammars, (ii) the meta-theoretical foundations in rewriting logic implemented in Maude [7] where arbitrary models can be transformed into their meta-representation; and (iii) object-based reflective programming languages such as Smalltalk. However, we believe that our proposal fits much better to the MDA framework for generic and meta-transformations due to its intuitive visual representation and relatedness to existing MDA standards.

The distinction between platform-independent and platform-specific transformation (see Sec. 5) was also facilitated in [4]. Here the authors propose the use of UML as the platform-independent language for designing transformations. In our approach, the PIT language is a combination of two formal notations (abstract state machines and graph transformation) but naturally, the mathematical background is hidden by using a UML-like visual syntax for transformation rules.

7 Conclusions

In the current paper, we proposed the use of *generic* and *meta-transformations* for solving transformation engineering problems within the MDA. Generic transformations (that allow the use of type (higher-order) variables in transformation rules) may capture transformation problems involving several metalevels in a very general, *compact* and platform-independent way. Meta-transformations

(that are transformations operating on transformations) offer support for increasing the *performance* of generic transformations by tailoring them into different platforms (modeling languages). Furthermore, the *maintenence* of models and transformation can also be carried out in a uniform way by meta-transformations.

The two main prerequisites for such a high-level transformation framework are (i) the explicit representation of instance-of relations in the modeling space, and (ii) the handling of transformation rules as models (as data). In the paper, we used the dynamic and multilevel VPM metamodeling framework [24], which provides support for both of these features. In fact, VPM also supports the *reusability* of transformation rules by introducing rule inheritance. However, we disregarded this very important software engineering aspect in the current paper to clearly distinguish our new results from any previous work.

Acknowledgments. This paper was influenced by several fruitful discussions at the Dagstuhl Seminar on Language Engineering in the Model Driven Architecture (March 2004). We are grateful to many of the participants and organizers.

References

1. The Eclipse project. `www.eclipse.org`.
2. D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth Intern. Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 243–258. Springer, Dresden, Germany, 2002.
3. C. Atkinson and T. Kühne. The essence of multilevel metamodelling. In M. Gogolla and C. Kobryn (eds.), *Proc. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts and Tools*, vol. 2185 of *LNCS*, pp. 19–33. Springer, 2001.
4. J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective model driven engineering. In P. Stevens, J. Whittle, and G. Booch (eds.), *Proc. UML 2003: 6th Intern. Conference on the Unified Modeling Language*, vol. 2863 of *LNCS*, pp. 175–189. Springer, San Francisco, CA, USA, 2003.
5. E. Börger and R. Stärk. *Abstract State Machines. A method for High-Level System Design and Analysis.* Springer, 2003.
6. T. Clark, A. Evans, and S. Kent. The Metamodelling Language Calculus: Foundation semantics for UML. In H. Hussmann (ed.), *Proc. Fundamental Approaches to Software Engineering, FASE 2001 Genova, Italy*, vol. 2029 of *LNCS*, pp. 17–31. Springer, 2001.
7. M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming.* CSLI Publications, Stanford University, 2000.
8. J. de Lara and H. Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber (eds.), *5th Intern. Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings*, vol. 2306 of *LNCS*, pp. 174–188. Springer, 2002.
9. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.

10. G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic (eds.), *UML 2000 - The Unified Modeling Language. Advancing the Standard*, vol. 1939 of *LNCS*, pp. 323–337. Springer, 2000.
11. C. Ermel, M. Rudolf, and G. Taentzer. *In [9]*, chap. The AGG-Approach: Language and Tool Environment, pp. 551–603. World Scientific, 1999.
12. T. Gardner, C. Griffin, J. Koehler, and R. Hauser. A review of OMG MOF 2.0 Query / Views / Transformations submissions and recommendations towards the final standard. In *Workshop on Metamodeling for MDA*, pp. 179–197. 2003.
13. J. H. Hausmann and S. Kent. Visualizing model mappings in UML. In *SoftVis 03: ACM Symp. on Software Visualization*, pp. 169–178. San Diego, CA, USA, 2003.
14. R. Heckel, J. Küster, and G. Taentzer. Towards automatic translation of UML models into semantic domains. In *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pp. 11–21. Grenoble, France, 2002.
15. W. Hesse. Two-level graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg (eds.), *Intern. Workshop on Graph-Grammars and Their Application to Computer Science and Biology, October 30 - November 3, 1978*, vol. 73 of *Lecture Notes in Computer Science*, pp. 255–269. Springer, Bad Honnef, 1979.
16. D. Milicev. Automatic model transformations using extended UML object diagrams in modeling environments. *IEEE Transactions on Software Engineering*, vol. 28(4):pp. 413–431, 2002.
17. Object Management Group. *QVT: Request for Proposal for Queries, Views and Transformations*. `http://www.omg.org`.
18. Object Management Group. *XML Metadata Interchange*. `http://www.omg.org/technology/documents/formal/xmi.htm`.
19. Object Management Group. *Meta Object Facility Version 2.0*, 2003. `http://www.omg.org`.
20. I. Porres. Model refactorings as rule-based update transformations. In P. Stevens, J. Whittle, and G. Booch (eds.), *Proc. UML 2003: 6th Intern. Conference on the Unified Modeling Language*, vol. 2863 of *LNCS*, pp. 159–174. Springer, San Francisco, CA, USA, 2003.
21. QVT-Partners. Revised submission for MOF 2.0 Query / Views / Transformations RFP, 2003. `http://qvtp.org`.
22. A. Schürr, A. J. Winter, and A. Zündorf. *In [9]*, chap. The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.
23. J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, and G. Karsai. Domain translation using graph transformations. In *Proc. Tenth IEEE Intern. Conference and Workshop on the Engineering of Computer-Based Systems*, pp. 159–168. Huntsville, AL, 2003.
24. D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, vol. 2(3):pp. 187–210, 2003.
25. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, vol. 44(2):pp. 205–227, 2002.
26. J. Whittle. Transformations and software modeling languages: Automating transformations in UML. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth Intern. Conf. on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 227–242. Springer, Dresden, Germany, 2002.