Customizing GrGen.NET for Model Transformation

Tom Gelhausen, Bugra Derre, and Rubino Geiss
Institute for Program Structures and Data Organization (IPD)
University of Karlsruhe (TH)
Karlsruhe, Germany
gelhausen@ipd.uka.de, s_derre@ira.uka.de, rubino@ipd.info.uni-karlsruhe.de

ABSTRACT

We discuss a method for customizing general-purpose graph rewriting systems for the task of model transformation. The portable approach is as simple as effective: It offers full static UML compliance including all diagram types for typed graph rewriting systems providing textual interfaces. We exemplify our approach by means of our implementation.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability—*Data mapping*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*

General Terms

Design, Languages

Keywords

Graph Rewriting, Model-to-Model, Transformation, MOF, UML, XMI, Mapping

1. INTRODUCTION

General-purpose graph rewriting systems (GRS) provide powerful and just as easy-to-handle operations on complex data structures; recent research led to a remarkable speed in execution [19]. Yet, as these systems are general-purpose, they are per se futile: They need customization on the operational area, i. e. an adequate model definition. This paper describes a method to automatically obtain the model definition for UML structures for the general-purpose graph rewrite system GrGen.NET [9]. Furthermore, we present a matching pair of import and export filters [11] for UML models stored in the standard XMI format [16].

Several publications state that the execution speed of the model transformations is crucial for an integrated development process [5, 6]. GrGen.NET is one of the fastest GRS available [10, 19]. Furthermore, its expressiveness is at least competitive among other GRS and thus surpasses QVT [17] — currently the most prominent model transformation approach — with respect to its pattern matching and rewriting capabilities[1]. New features like dynamic patterns in the upcoming 2.0 release of GrGen.NET will even extend the lead. All these aspects render GrGen.NET a more than adequate basis for model transformation.

The method we propose is straight forward: The UML specification is available in XMI and we provide XSLT scripts to transform it into model definition files for GrGen.NET. The purpose of this paper is to a) enable the users of our model definition to grasp how to use this model (i. e. predict the names of the node and edge types easily), and b) enable the authors of other GRS to also create UML-compliant model definitions. The latter has the prospect of simplifying the exchange of UML data among different GRS via the already-standard GXL [12] without the need of writing import and export filters for XMI. This way, a comparison of the true model transformation capabilities of the various GRS would come into reach.

1.1 Metamodeling

Our understanding of model transformation is primarily influenced by the Object Management Group's (OMG) Model Driven Architecture (MDA). Consequently, we consider the structures of UML as the basic vocabulary for the expression of models, not only class diagrams like the majority of tools. Fortunately, the comprehensive UML specification is available in machine-readable form [14] — more precisely in a form that enables us to automatically derive a model definition. Unfortunately, the OMG's opus comes with a respectable bouquet of other specifications and terms and definitions that render the task somewhat confusing [13, 14, 16, 17, 18]. We cannot reproduce all of their content in this paper for it to be self-contained; our publication should rather serve to untangle things. In particular, it should not be necessary to read additional specifications to understand what we discuss in this paper.

At this point, we need to introduce the MOF metalevels M0 to M3. Understanding this concept is important to keep track through the rest of this paper. If you are familiar with these metalevels, you may advance to the next section.

The OMG, publisher of the UML standard, developed the MetaObject Facility (MOF) to specify the elements of UML (and other things) and their semantics. Comparable with

[1]In particular, no complete implementation for QVT is known to the authors.

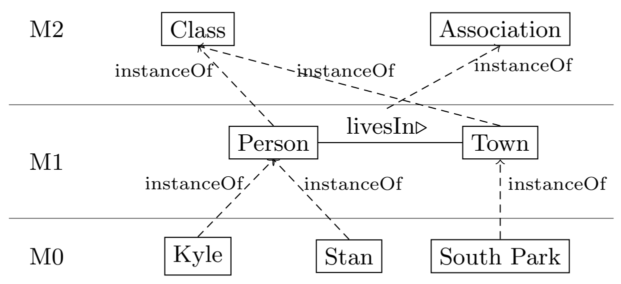


Figure 1: Example for metalevels

the foundation of graph theory in mathematics (e.g. category theory), UML is founded in MOF. MOF, a kind of lightweight UML, is founded in itself — just like mathematics. Concerning "definition" of things available in a universe (i.e. modeling), we can build up a hierarchy: If we regard the things utilized to define the model M for a universe U as another universe U', we can define a model M' for U'. M' defines legal structures within the universe U'. We call this model M' a metamodel. In total, the OMG defines four so called metalevels M0, M1, M2, and M3.

Example: The level M0 defines the universe of things in our world that we want to manage with our system. Let "Kyle and Stan live in South Park" be our universe U. The level M1 defines classifications for what is available in U, i.e. a class Person and a class Town and an association livesIn. Kyle and Stan are instances of Person, and South Park is an instance of Town. M1 defines the model of our universe. The level M2 defines classifications for the level M1: The level M1 may contain Classes and Associations. Person is an instance of Class and livesIn is an instance of Association. M2 defines the metamodel. Figure 1 illustrates this example.

The attentive reader of the preceding example may have noticed that we did not introduce M3 here. This is because we do not need M3 to explain our mappings. Section 3 goes into further details.

1.2 Mappings

Regarding an ordinary software project, we find the structures we want to manipulate with model transformation techniques on the level M1 (symbolized by ② in Figure 2)[2]. Yet for our GRS, this *is* the effective data it shall operate on — from the GRS' point of view it is M0-data ④. So we need transformation scripts to map the contents from a UML diagram ② into node- and edge-creating directives for our GRS ④. Logically — and for strictly typed GRS like GrGen.NET also formally — we need a model definition for the node- and edge-types we want to use. This model definition is level M1 in the terms of the OMG. We obtain it by mapping M2 from the UML metamodel "stack" ① to M1 in the metamodel stack on the GrGen.NET side ③.

2. EXAMPLE

To give an impression of the resulting graph model we present an exemplary model transformation: Assume we want to eliminate ϵ-transitions from a state machine we modeled in a UML tool. Therefor we load the XMI representation of the state machine into our GRS. By applying several graph transformation rules the GRS produces an ϵ-transition free state machine. The last step is to export the resulting graph to an XMI file again.

[2]We use the circled numbers throughout this document to clarify which "model" we are currently referring to.

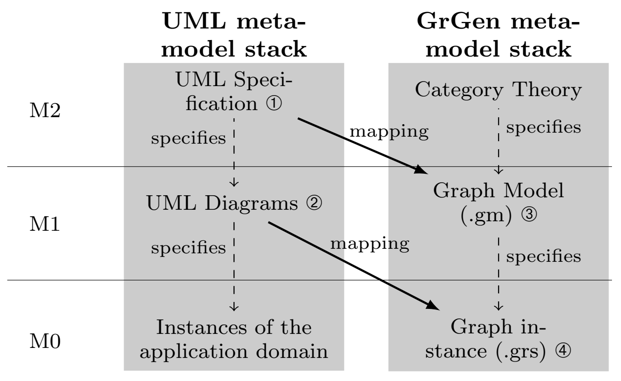


Figure 2: Illustration of the mappings

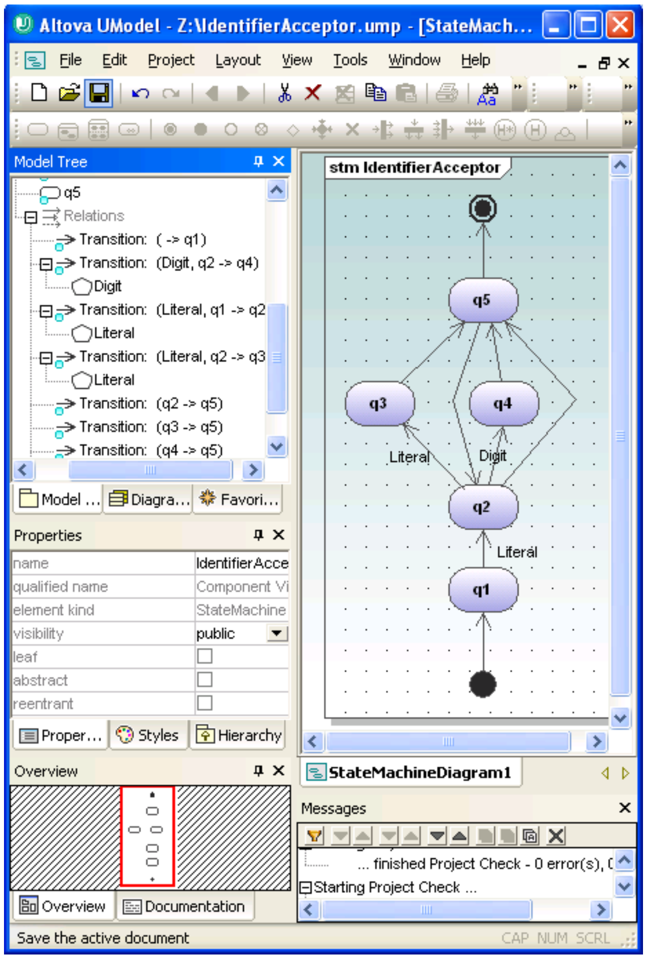
One of the rules of the above transformation is shown in Listing 1. It deletes transitions that do not have a trigger, i.e. are ϵ-transitions. The first line of the rule declares a parameter sm. When applying this rule, we pass the state machine we currently work on via this parameter. Line 2 navigates to a contained region and line 3 to a transition within this region. Note that the names of the elements conform precisely to those of the UML specification [18]: The type names we used in the rule are highlighted in Figure 5, showing an excerpt of the UML Superstructure specification of state machines. The negative application condition (NAC) in the lines 4 to 8 excludes transitions from a start node, i.e. a transition whose source is a pseudo state of the kind "initial". The second NAC in line 9 requires that the transition trans shall not have any trigger (the dot matches any node). If all these conditions are true trans needs to be an ϵ-transition. Line 10 contains the command to delete the transition trans.

The rule removeEpsilonTransition obviously cannot do the work on its own. It is applied after other transformation steps introduced the necessary transitive transitions and is iterated until it does not match any more.

Figure 3 shows a finite acceptor for identifiers modeled as state machine in Altova UModel 2008 [2]: In the subwindow "Model Tree", ϵ-transitions do *not* contain a child (depicted with a pentagon), whereas the graphical repre-

```
 1 rule removeEpsilonTransition(sm:StateMachine) {
 2   sm -:region-> reg:Region;
 3   reg -:transition-> trans:Transition;
 4   negative {
 5     trans -:source-> initialState:Pseudostate;
 6     initialState -:kind-> psk:PseudostateKind;
 7     if { psk.value = ENUM_PseudostateKind::initial; }
 8   }
 9   negative { trans -:trigger-> .; }
10   modify { delete(trans); }
11 }
```

Listing 1: Transformation rule working on the graph representation of a state machine


Altova UModel - Z:\IdentifierAcceptor.ump - [StateMach...

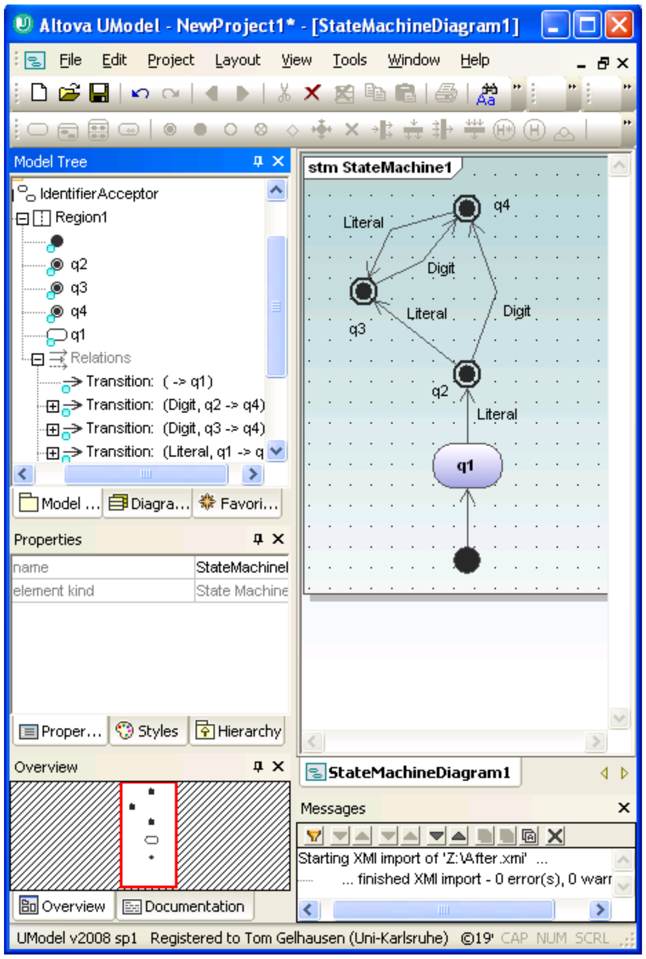
Model Tree
- q5
- Relations
  - Transition: (-> q1)
  - Transition: (Digit, q2 -> q4)
    - Digit
  - Transition: (Literal, q1 -> q2
    - Literal
  - Transition: (Literal, q2 -> q3
    - Literal
  - Transition: (q2 -> q5)
  - Transition: (q3 -> q5)
  - Transition: (q4 -> q5)

Properties

name	IdentifierAcce
qualified name	Component Vi
element kind	StateMachine
visibility	public
leaf	☐
abstract	☐
reentrant	☐

Messages
```
... finished Project Check - 0 error(s), (
Starting Project Check ...
```
Save the active document

Figure 3: A finite acceptor as a state machine


Altova UModel - NewProject1* - [StateMachineDiagram1]

Model Tree
- IdentifierAcceptor
  - Region1
    - q2
    - q3
    - q4
    - q1
  - Relations
    - Transition: (-> q1)
    - Transition: (Digit, q2 -> q4)
    - Transition: (Digit, q3 -> q4)
    - Transition: (Literal, q1 -> q

Properties

name	StateMachinel
element kind	State Machine

Messages
```
Starting XMI import of 'Z:\After.xmi' ...
... finished XMI import - 0 error(s), 0 warr
```
UModel v2008 sp1 Registered to Tom Gelhausen (Uni-Karlsruhe) ©19

Figure 4: The state machine without ϵ-transitions

sentation omits edge labels in those cases. We exported this machine to an XMI file, imported it into GrGen.NET, transformed it with the sketched rule set, exported it to an XMI file again, loaded this XMI file into a new project in Altova UModel 2008, and imported the model elements into a new state machine *diagram*. (The layout is not kept during the round trip, because XMI has no standardized means of layout information.) The result can be seen in Figure 4.

3. IMPLEMENTATION

The main challenge of transforming the UML specification ① into a GrGen.NET model ③ and a UML model ② into a graph definition ④ is the mapping of the available primitives. The reverse mapping, the export of a UML graph ④ to an XMI document ②, is also described here.

3.1 Compliance

The OMG compliance levels ("L0" to "L3", and "LM") specify which subset of all available structures and features of UML are allowed in a certain application. We chose "L3.merged.cmof" [14] as the source file for the generation of our UML graph model as it represents the complete UML (cf. [15]). Moreover in L3, all structures are in one single package avoiding mutual imports and exports of different elements.

3.2 Creating the Model

The OMG uses CMOF [13], the "Complete MOF", to specify UML [18]. In terms of the metamodel stack, CMOF is the meta-metalevel M3. As expressing the UML specification does not require all modeling elements of the CMOF specification, we only need to map elements actually used. These modeling elements are listed in Table 1, together with their mapping to GrGen.NET constructs.

CMOF	**GrGen.NET** (③, .gm-file)
CMOFAssociation	–
CMOFClass	`node class`
CMOFComment	*// comment*
CMOFConstraint	–
CMOFEnumeration	`enum`
CMOFEnumerationLiteral	`enum::item`
CMOFOpaqueExpression	–
CMOFOperation	–
CMOFParameter	–
CMOFPrimitiveType	wrapped in `node class`
CMOFProperty	`edge class`

Table 1: CMOF structures used to define UML

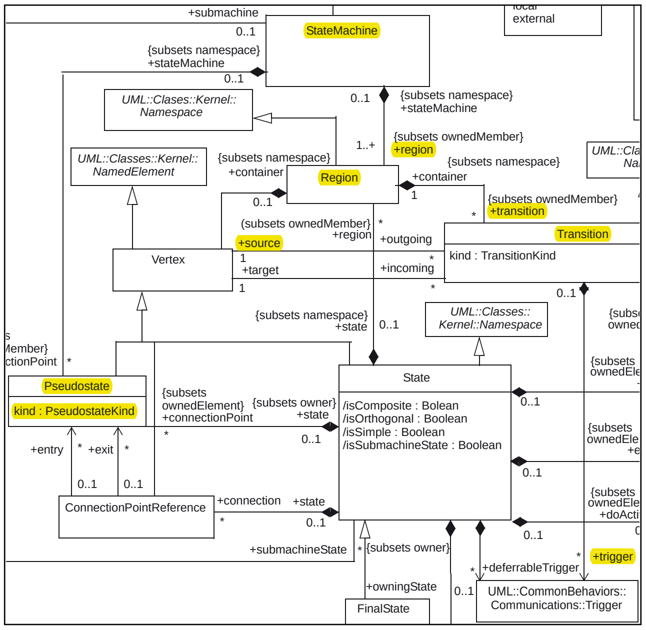
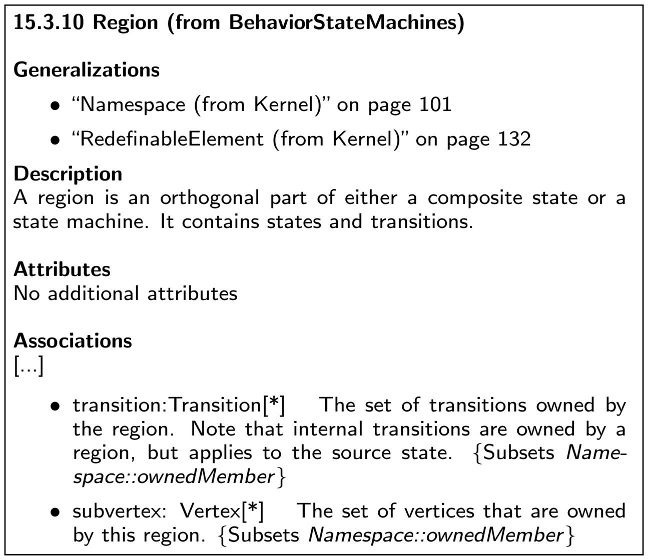


Figure 5: Excerpt from "Figure 15.2 – State Machines" of the UML Superstructure specification



15.3.10 Region (from BehaviorStateMachines)

Generalizations

- "Namespace (from Kernel)" on page 101
- "RedefinableElement (from Kernel)" on page 132

Description
A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.

Attributes
No additional attributes

Associations
[...]

- transition:Transition[*] The set of transitions owned by the region. Note that internal transitions are owned by a region, but applies to the source state. {Subsets *Namespace::ownedMember*}
- subvertex: Vertex[*] The set of vertices that are owned by this region. {Subsets *Namespace::ownedMember*}

Figure 6: Excerpt from the UML Superstructure specification concerning Regions

3.2.1 Mapping

We map CMOFClasses to node classes. The name of the node class is the name of the corresponding UML element. Of course, we do not expect the user to obtain these names from the UML specification expressed in CMOF, i.e. an XMI document. But they exactly correspond to the names used in the (human readable) Superstructure specification (cf. the excerpt in Figure 6). This is possible at all only due to the names of all CMOFClasses being unique throughout L3, leading to elegant naming conventions.

A CMOFProperty defines a name for a value in the context of a CMOFClass (comparable to an attribute in an object-oriented language). This value is usually embodied by another CMOFClass representing the actual property value. A CMOFProperty may be optional or may exist multiple times in a given instance, so representing it by a GrGen.NET class attribute is not possible. Hence, we map CMOFPropertys to edge classes pointing from the owning type to the value type.

Unfortunately, the name of CMOFPropertys is not unique as it is the case for CMOFClasses. This necessitates the aggregation of possible incident types for the connection assertions of those edges. This way, we can use the simple name of a property instead of a bulky unique identifier. The downside of this approach is that it leads to slightly more overhead when accessing reflection information (cf. Section 3.3).

Each CMOFPrimitiveType is represented by a node class. These node classes share a common abstract super type defining an abstract attribute (i.e. an attribute name declaration without a type declaration) named value. Each concrete CMOFPrimitiveType implements this attribute with a corresponding type from the available primitive types of GrGen.NET. CMOFEnumerations are mapped to `enums` and CMOFEnumerationLiterals to the according `enum` items. In addition, `enums` are treated like CMOFPrimitiveType types.

We currently do not support dynamic behavior of UML (or other things specified in CMOF). So we do not map CMOFConstraints, CMOFOpaqueExpressions, CMOFOperations, and CMOFParameters to first class citizens. Nevertheless, these constructs are available via our reflection mechanism (cf. Section 3.3). This way, support for the OCL expressions in CMOFConstraints or CMOFOpaqueExpressions seems feasible via a graph-rewrite-based interpreter. The same is true for CMOFOperations and their CMOFParameters.

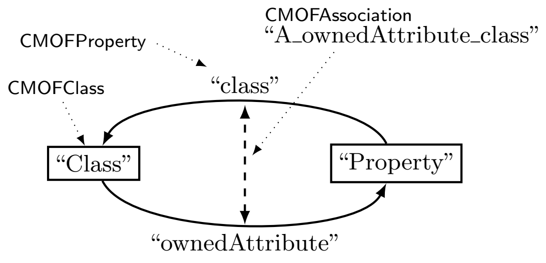


Figure 7: The Role of CMOFAssociations

To support CMOFAssociations, one would need an expressiveness as it is provided in the Omnigraph Rewrite System (Ogre) [8]: CMOFAssociation are binary relations between CMOFPropertys (see Figure 7). This leads to omniedges representing associations that connect other omniedges representing properties. Yet in contradiction to what one might expect, CMOFAssociations do not prescribe any further actions if they are inserted or deleted. Due to the additional complexity introduced by the Ogre tool stack, we refrain from supporting them. CMOFComments are mapped to comments in the model definition syntax of GrGen.NET.

Note that our approach can very well handle associations in UML models, because these are instances of a CMOFClass named "Association"; in contrast CMOFAssociation are entities on a higher meta-level. We want to emphasize that the supported CMOF structures effectively suffice to express all current UML diagram types with all of their elements.

```
 1 // A region is an orthogonal part of either a composite state
 2 // or a state machine. It contains states and transitions.
 3 node class Region extends RedefinableElement, Namespace {
 4    uuid = "Region";
 5 }
 6
 7 edge class subvertex extends CMOFProperty
 8    connect Region[0:*] -> Vertex[*] {
 9       name = "subvertex";
10 }
11
12 edge class transition extends CMOFProperty
13    connect Region[0:*] -> Transition[*] {
14       name = "transition";
15 }
```

Listing 2: An excerpt from the graph model definition generated by our scripts

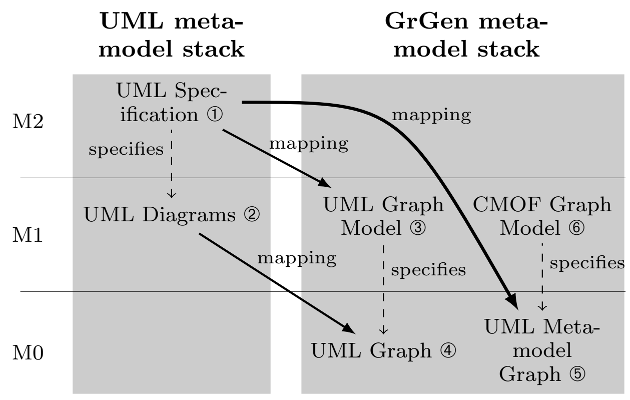


Figure 8: Metamodel Graph

3.2.2 *Example*

Listing 2 shows an excerpt from the generated model definition file. It declares elements used in the state machine example (cf. Listing 1). The excerpt starts with a comment that can directly be copied from the CMOF document. Line 3 defines a class of nodes representing regions of state machines. A Region inherits from RedefinableElement and Namespace. The following line declares a node attribute uuid. This attribute is initialized with the string value "Region" for every instance of this node class: We made *all* node classes contain a uuid attribute to support reflection. We can use its value to link nodes to their corresponding metadata (cf. Section 3.3).

The lines 7 and 12 declare edge classes representing the properties subvertex and transition of a Region. The property subvertex can store arbitrarily many Vertexes, the transition property arbitrarily many Transitions. Both edge class declarations contain a name attribute for reflection support.

3.3 Reflection Information

The XMI importer and exporter (and certainly other applications) require metadata about the current graph (cf. ④ in Figure 8 which extends Figure 2). As GrGen.NET does not offer reflection information besides typeof expressions, we import the UML specification ① itself as a graph instance ⑤. This graph instance adheres to a model ⑥ that represents *all* CMOF constituents as node classes. The attributes of these constituents are represented the same way as in the UML graph model ③ (cf. Section 3.2), but this time, we only allow strings as primitive types. While importing L3 into a graph instance ⑤, we can collect node and edge types necessary for representing this graph. This information becomes the metamodel ⑥. So we "distill" this model from the CMOF specification of L3.

The metamodel graph ⑤ contains nodes representing C-MOF constituents. These nodes have an attribute uuid containing a unique string. All nodes of the actual graph ④ also have a uuid attribute. They each share their uuid-values with a node of the metamodel graph ⑤. Via a natural join, the desired information can be obtained. The reasons not to express this relationship with edges were a) independence of the models ③ and ⑥ and b) that new instances of arbitrary classes automatically refer to the correct metamodel element by using GrGen.NET's default attribute initializer.

The metadata for edges is obtained likewise. Yet, instead of unique identifiers for the graph elements ④ we have possibly ambiguous names (cf. 3.2). Hence we have to find the right metamodel element using information about the owning types in the graphs ④ and ⑤.

3.4 Reading XMI

Our implementation of the XMI import (as well as the export) follows chapter 6 ("XML Document Production") of the MOF/XMI mapping specification [16].[3] Hence, our implementation can import every XMI representation[4]: The precondition is that the related node and edge types exist, e. g. the graph model for UML diagrams ③ in our case. If a document type is specified using CMOF, the model can be generated as presented in this paper. In this respect, the presented import and export procedures are generic and not limited to UML documents.

According to the production scheme, the XML node names as well as the XML attributes from the default namespace represent properties and thus create edges. The type of these edges corresponds to the name of the XML element or the XML attribute, respectively. For the example in Listing 4, these properties are region, name, subvertex, outgoing, transition, source, target, and incoming.

The nodes that are to be connected by these edges represent mostly CMOFClasses: Their type and thus the name of the node class can be obtained from the type attribute from the XMI namespace (i.e. xmi:type in Listing 4). For the given excerpt, these are Region, Transition, and Pseudostate.

A value that is not a CMOFClass is either an elementary value, an enumeration value, or a reference value. We obtain the information about the type of the value from a lookup in the metamodel. Elementary and enumeration values are wrapped in nodes of the appropriate type (cf. Section 3.2). Otherwise we have to resolve a reference value: All instances of CMOFClasses have an id attribute in the XMI namespace (e. g. xmi:id="f900" in line 5 of Listing 4). These identifiers can be referenced in XMI either by idref attributes in the XMI namespace (e. g. xmi:idref="f900" in line 3), or by at-

[3]In fact, section 6.5.2 ("EMOF Package") together with section 6.4 ("EBNF Rules Representation") are sufficient for our task: These two sections describe how to parse as well as how to unparse XMI documents.
[4]If it is compliant with version 2.1 of XMI.

```
 1 <!-- This template links the parent-node to its child-nodes
 2      with its corresponding edge type. -->
 3 <xsl:template name="link-parentNode-with-childNodes">
 4 <xsl:param name="parent-node" />
 5 <xsl:param name="child-nodes" />
 6   <!-- For each NON-empty child node ... -->
 7   <xsl:for-each select="$child-nodes[@xmi:type]">
 8     <!-- ... create the child node, ... -->
 9     <xsl:text>new </xsl:text>
10     <xsl:value-of select="@xmi:id" />
11     <xsl:text> : </xsl:text>
12     <xsl:value-of select="substring(@xmi:type,5)" />
13     <xsl:text>()</xsl:text>
14     <!-- ... create the edge ... -->
15     <xsl:text>new </xsl:text>
16     <!-- ... beginning at the parent ... -->
17     <xsl:value-of select="$parent-node/@xmi:id"/>
18     <xsl:text> -: </xsl:text>
19     <!-- ... with the according edge type ... -->
20     <xsl:value-of select="name()"/>
21     <!-- ... referencing its child  (actual context) node -->
22     <xsl:text> -> </xsl:text>
23     <xsl:value-of select="@xmi:id"/>
24     <!-- Apply this template recursively to the child
25          element, if no more children are found the
26          recursive application of the template will stop -->
27     <xsl:call-template name="link-parentNode-with-childNodes">
28       <xsl:with-param name="parent-node" select="."/>
29       <xsl:with-param name="child-nodes"
30         select="./child::node()"/>
31     </xsl:call-template>
32   </xsl:for-each>
33 </xsl:template>
```

Listing 3: XSLT excerpt linking parent to children

```
 1 <region xmi:type="uml:Region" xmi:id="8709" name="Region1">
 2    <subvertex xmi:type="uml:Pseudostate" xmi:id="9f1a">
 3       <outgoing xmi:idref="f900" />
 4    </subvertex>
 5    <transition xmi:type="uml:Transition" xmi:id="f900"
 6       source="9f1a" target="7e93" />
 7    <subvertex xmi:type="uml:Pseudostate" xmi:id="7e93">
 8       <incoming xmi:idref="f900" />
 9    </subvertex>
10 </region>
```

Listing 4: An excerpt from an XMI document generated by Altova UModel 2008

```
 1 // Write the properties of the class node
 2 rule writeProperties(parent:CMOFClass) {
 3    parent -prop:CMOFProperty-> child:CMOF_NODE;
 4    modify {
 5       emit("\t<",  prop.uuid, "␣xmi:type=", "\"uml:");
 6       emit(child.uuid, "\"", "␣xmi:id=\"G-", child.id, "\"");
 7       exec(writeAttributeWithValueReference(child)[*]);
 8       exec([writeAttributeWithIdReference(child)]);
 9       emit(">\n");
10       exec(writeProperties(child)[*]);
11       exec(writeXmiIdReference(child)[*]);
12       emit("\t</",  prop.uuid,">\n");
13       delete(prop);
14    }
15 }
```

Listing 5: An excerpt from the rewrite rules generating XMI

tributes that directly reference them (e. g. target="7e93" in line 6 refers to the Pseudostate in line 7).

We implemented this mapping in XSLT (cf. Listing 3). The script process the whole graph in a recursive fashion (see line 27) until we encounter leafs.

3.5 Writing XMI

Following the XMI production scheme (cf. Section 3.4), the export code becomes straight forward, when reflection information is at hand (cf. Section 3.3). We recursively traverse the UML graph ④ beginning at a dedicated start node. While generating XMI the traversed properties are deleted to break possible cycles and thus ensure termination.

Listing 5 shows an excerpt from the rule set generating XMI out of a graph instance ④. It is applied to instances of CMOFClasses and matches if the class has properties. This is ensured by line 3. The lines 5 and 6 print the opening XML tag with two attributes, xmi:type and xmi:id. Lines 7 and 8 produce all XML attributes for the parent node regarding the according child, whereas the lines 10 and 11 produce all XML child elements. Note the recursive call of writeProperties in line 10 ensuring the traversal of the complete graph. Line 12 closes the XML tag. Finally, the property edge is removed.

4. RELATED WORK

QVT (Queries/Views/Transformations) [17] is probably the most prominent approach to map source models to target models and transform these models by applied pattern matching. It uses a hybrid declarative/imperative style of specification. The metamodel and query language of QVT support object pattern matching and object template creation. QVT implements some kind of triple graph grammars: “Traces” are links between elements of (potentially) different models. These traces are automatically maintained during transformations. They enable the propagation of changes in either model to the other model. A full implementation of QVT provides mapping and transformation capabilities within one single framework (cf. [4]). However, QVT is just a specification. To the present day no fully compliant implementation is available; existing implementations vary significantly in the supported features. In particular, an implementation for the UML metamodel is needed for usage in queries and transformations. In this respect QVT is as suitable or unsuitable for the task as GrGen.NET or any other GRS. Moreover it is debated in the model transformation community whether the QVT languages are adequate at all.

GREAT [5] (not GReAT, see below) is a rule-based transformation framework which facilitates mapping transformations on the same or on different abstraction levels. It can automate model transformation tasks such as refactorings, model analysis, or design pattern application. The graph model transformations are processed by the graph rewrite system OPTIMIX [3]. Yet, with its predefined and limited graph model, it focuses on a subset of UML class diagrams only. A need for improved performance and usability is mentioned in [5]. Like our approach, GREAT provides roundtrip engineering for the XMI standard.

GReAT [1] is a graph rewrite system specifically designed to enable the rapid development of domain specific languages (DSL). It consists of a generic modeling environment (GME) enabling the development of a mixed visual and textual notations for DSLs. GReAT itself is capable of transforming the graph-based representations of those languages. The export of these representations into textual form is only supported by means of special visitors (design pattern) that have to be written in C++. In contrast, GrGen.NET seamlessly integrates the production of the output into the rule language. Comparisons [21] show that the mixed visual and textual modeling language is quite verbose, while GReAT's execution speed is rather low. Most notably, no DSL for UML with comparable coverage is available for GReAT to the knowledge of the authors. Thus our approach could provide a solution for GReAT, too.

VIATRA2 [22] is a GRS especially designed for model transformation. Its execution semantics is based on a reduction to abstract state machines (ASMs). The meta-model for UML is derived — similar to our approach — by using MOF specifications [23]. Considering the input filters, VIATRA2 uses an ad-hoc implementation for UML. Our approach is more general and not restricted to only UML. VIATRA2 generates text by using nodes and edges to simulate monadic output. We observed that this approach can be tedious [7]. Instead, we use imperative features of GrGen.NET (namely the `emit` and `exec` statement of the rewrite part. The transformations themselves benefit from GrGen.NET's imperative features, too, because we are able to specify warning and error messages in a way familiar to the rule developer.

VIATRA2 supports a high level of reflective (meta-) rule manipulation, but therefore lacks performance: The potentially evolving rules cannot be mapped to efficient implementations easily. Performance analysis done by the tool's authors [6] and by an independent test case [19] suggest that substantial transformations done on non-trivial models will take too long to be integrated smoothly into the software development process.

EMF Tiger (EMT) [20] is an Eclipse EMF/GMF based visual editor generator for AGG. It is built upon the Eclipse Modeling Framework (EMF) which creates Java classes from XMI documents. EMT is capable of reading XMI files, but the only model used to represent elements of this file in a graph is EMOF, a subset of CMOF. In this way, it is possible to read the UML specification and transform *this* specification into something else. It is also possible to read a UML model stored in an XMI file and to transform it. However, there is no connection between the model and its metamodel. Consequently, this approach inherently lacks type safety. Like GReAT and VIATRA2, EMT's performance is unsatisfactory for complex transformations tasks.

5. CONCLUSION

General-purpose graph rewrite systems like GrGen.NET require customization on the application domain. In the case of model transformation, we need a customized graph model corresponding to the UML metamodel. As the UML specification is available in XMI format, we wrote an XSLT script automatically generating the required graph model definition.

In this paper, we additionally presented XMI import and export filters, that enable the user to exchange the transformed models with the "real world". We demonstrated the success of this approach by a round trip from and to the commercial UML modeling tool Altova UModel 2008. This task was significantly simplified by our approach for storing reflection information. The resulting scripts are not limited to process UML documents.

The main challenge of this work is not to get lost in the details: The primary objectives of this paper are to help our readers to really understand the model transformation process along with our extensions for GrGen.NET [11]. This paper also gives a profound description for achieving support for the static contents of all UML diagrams with all of their elements. Most notably, this can be done by implementing a marginal amount of code.

Acknowledgments..

We want to thank Andreas Höfer, Sven Körner, Mathias Landhäußer, Jakob Blomer, Sebastian Buchwald, and Edgar Jakumeit for proofreading the paper. We are particularly grateful for the help of the students who implemented GrGen.NET: Moritz Kroll, Sebastian Buchwald, Edgar Jakumeit, and others. We also want to thank the anonymous reviewers for the helpful critique and comments.

6. REFERENCES

[1] A. Agrawal. Metamodel Based Model Transformation Language. In R. Crocker and G. L. S. Jr, editors, *OOPSLA Companion*, pages 386–387. ACM, 2003.

[2] Altova. UModel – UML Tool for Software Modeling and Application Development. `http://www.altova.com/products/umodel/uml_tool.html`, Jan. 2008.

[3] U. Assmann. OPTIMIX – a tool for rewriting and optimizing programs. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*, pages 307–318, 1999.

[4] G. Caplat and J. L. Sourrouille. Considerations about Model Mapping. In *WiSME@UML'2003*, San Francisco, USA, Oct. 2003. Workshop in Software Model Engineering (WiSME@UML'2003).

[5] A. Christoph. Describing Horizontal Model Transformations with Graph Rewriting Rules. In U. Aßmann, M. Aksit, and A. Rensink, editors, *MDAFA*, Volume 3599 of *LNCS*, pages 93–107. Springer, 2004.

[6] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA – Visual Automated Transformations for Formal Verification and Validation of UML Models. In *Proc. 17th IEEE International Conference on Automated Software Engineering ASE 2002*, pages 267–270, Sept. 2002.

[7] O. Denninger. Erweiterung des Kantenkonzepts deklarativer Graphersetzungssysteme von Einfachkanten über Hyperkanten zu „Superkanten". Diplomarbeit, Universität Karlsruhe, IPD Tichy, 2007.

[8] O. Denninger, T. Gelhausen, and R. Geiß. Applications and Rewriting of Omnigraphs – Exemplified in the Domain of MDD. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, *LNCS*. Springer, 2008.

[9] R. Geiß. GrGen.NET. http://www.grgen.net/, Jan. 2008.

[10] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski. Grgen: A Fast SPO-Based Graph Rewriting Tool. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *ICGT*, Volume 4178 of *LNCS*, pages 383–397. Springer, 2006.

[11] T. Gelhausen, B. Derre, and R. Geiß. The GrGen.NET MOF-Suite. http://www.grgen.net/mof, Jan. 2008.

[12] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In A. Winter, editor, *Seventh Working Conference on Reverse Engineering*, pages 162–171, 2000.

[13] OMG. *Meta Object Facility (MOF) Core Specification*. Object Management Group, Needham, MA, USA, Jan. 2006.

[14] OMG. *New XMI 2.1.1 Specification – UML 2.1.1 XMI file(s)*. Object Management Group, Needham, MA, USA, Oct. 2006.

[15] OMG. *Infrastructure Specification*. Object Management Group, Needham, MA, USA, 2.1.2 edition, Nov. 2007.

[16] OMG. *MOF 2.0/XMI Mapping, Version 2.1.1*. Object Management Group, Needham, MA, USA, Dec. 2007.

[17] OMG. *MOF Query/View/Transformation Specification*. Object Management Group, Needham, MA, USA, 2.0 edition, Jul. 2007.

[18] OMG. *Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. Object Management Group, Needham, MA, USA, Nov. 2007.

[19] A. Schürr, M. Nagl, and A. Zündorf, editors. *AGTIVE - Applications of Graph Transformation 2007*, volume Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) of *LNCS*. Springer, 2008.

[20] G. Taentzer. Tiger EMF Transformation. http://tfs.cs.tu-berlin.de/emftrans, 2007.

[21] G. Taenzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiß, A. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. Plump, and T. Vajk. Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07), LNCS*. Springer, 2008.

[22] D. Varró and A. Balogh. The Model Transformation Language of the VIATRA2 Framework. *Sci. Comput. Program.*, 68(3):187–207, 2007.

[23] D. Varró, G. Varró, and A. Pataricza. Designing the Automatic Transformation of Visual Languages. In *Sci. Comput. Program*, Volume 44, pages 205–227. Department of Measurement and Information Systems, Budapest University of Technology and Economics, Hungary, Aug. 2002.