Transformation of Models in(to) a Generic Metamodel

David Kensche Christoph Quix
Informatik V, RWTH Aachen University, 52056 Aachen , Germany
{kensche, quix}@cs.rwth-aachen.de

Abstract: *Model Management* aims at developing new technologies and mechanisms to support the integration, evolution and matching of models. Such tasks are to be performed by means of a set of *operators* which work on models and their elements. Furthermore, model management performs these operations generically, that is, without being restricted to a particular metamodel (e.g. the relational or XML Schema metamodel). In order to allow this, a generic metamodel must be used for model representation. Operators manipulate exclusively models described in that generic language. Consequently, models represented in concrete metamodels have to be imported into the generic metamodel and vice versa. In this paper we describe how we implemented rule based Import and Export operators between concrete metamodels and our generic role based metamodel *GeRoMe*. In addition, the same rule based approach can be used to implement one of the main model management operators, namely ModelGen, in a generic way. This operator is used to transform models using certain constructs into models using other modeling constructs.

1 Introduction

Research in *model management* aims at developing technologies and mechanisms to support the integration, merging, evolution, and matching of models. These problems have been addressed for specific modeling languages for a long time. Within an (integrated) information system, several metamodels are used, a specific one for each subsystem (e.g. DB system, application). Thus, the management of models in a generic way is necessary.

According to the IRDS standard [9], metamodels are *languages* to define models. Examples for metamodels are *XML Schema* or *UML*. The same terminology is adopted in the specifications of the Object Management Group (OMG, `http://www.omg.org`) for MOF (Meta Object Facility) and MDA (Model Driven Architecture). Models are the description of a concrete application domain. We will use the same terminology throughout this paper.

Generic model management supports the aforementioned tasks generically, that is, without being restricted to a particular modeling language [5, 6]. To achieve this goal, the definition of a set of *generic* structures representing models and the definition of *generic* operations on these structures are required [6]. These operations include, for instance, Merge which builds a mediated schema from two input models and a mapping in between [15]. Another example is ModelGen which transforms one model into another that is consistent with another modeling language.

Currently, model management applications often use graphs as generic data structure to represent models, but operators have to be aware of the employed metamodel [7, 8, 13]. While a graph representation is often sufficient for the purpose of finding correspondences between schemas (Match), it is not suitable for more complex operations (such as merging of models) as it does not contain detailed semantic information about relationships and

constraints. For example, in [14] a generic (but yet simple) metamodel is used that distinguishes between different types of associations in order to merge two models. Since we believe that simple graph based representations are not expressive enough for an integrated model management system that supports manipulation of models in general, we proposed the role based metamodel *GeRoMe* as a solution of this challenge [12].

In this paper we describe how we implemented Import and Export operators that transform schemas represented in concrete metamodels, such as the relational model, XML Schema, or OWL (Web Ontology Language), into equivalent models represented in our generic metamodel. Furthermore, we will apply the same approach to implement rule based set-at-a-time ModelGen operators which transform models between different modeling languages. Finally, we describe the additional requirements of these translation operators.

The rest of the paper is structured as follows. In section 1.1 we briefly describe our role based approach of generic metamodeling. Section 1.2 summarizes related work on translation operators. In section 2 we dwell on the challenges and implementation requirements of Import and Export operators whereas section 3 then gives details on our implementation of the aforementioned requirements. Section 4 compares the requirements of Import and Export operators to those of metamodel independent ModelGen operators. The last section summarizes our lessons learned from this work.

1.1 Our Metamodel: *GeRoMe*

Our *Ge*neric *Ro*le based *Me*tamodel *GeRoMe* (phonetic transcription: dʒerəʊm) [12] employs the *role based* modeling approach. In role based modeling, an object is regarded as playing roles in collaborations with other objects. Applied to generic metadata modeling this approach allows to *decorate* a model element with a combination of multiple predefined aspects, thereby describing the element's properties as accurately as possible while using only metaclasses and roles from a relatively small set. In such a generic metamodel, the different features of a model element (e.g. it is not only an *Aggregate* but also an *Association*) are only different views on the same element. During model transformations an element may gain or lose roles, thereby adding and revoking features.

Figure 1 shows part of the representation of an EER model for the airport domain and the equivalent representation in our generic metamodel *GeRoMe*. It contains a relationship type with an attribute and two entity types, the attributes of which we omitted from the

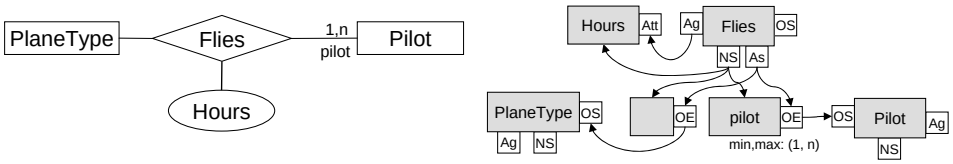

Figure 1: A small EER model and the *GeRoMe* representation of the same model

example. In *GeRoMe* each of the types, attributes and relationship ends is represented by an individual model element node (shown as grey rectangles). Every *GeRoMe* model element plays some roles depending on the features of the source model element that it represents. The Hours element plays an *Attribute* role (Att) as it represents an attribute, the entity types play ObjectSet roles (OS) since their instances have object identity (as opposed to the instances of a relational table) and *Aggregate* roles (Ag) as they are aggregates of attributes (not shown). The Namespace roles (NS) are played because the types define their attributes. The Flies element also plays the role of an *Association* which connects to two *ObjectAssociationEnds* (OE) (pilot and an anonymous one). It must be emphasized that this representation is not to be used by end users. Instead, it is employed internally by model management applications, with the goal to provide more information to operators than the usual graph based model.

1.2 Related Work

Whereas transformation of models between different modeling languages is a frequent task, it has up to now mainly been adressed in specialized settings which map from one particular metamodel to another fixed metamodel. The problem of *generic* model transformation has recently been adressed in [16] where models are represented as simple graphs and transformations are given as productions in the reserved graph grammar *RGG*.

Another approach to generic model representation has been introduced in [3], expressed in a relational model dictionary [1], and was recently used for the generic ModelGen implementation MIDST [2]. This approach differs from our representation in that it describes a class of model elements as a pattern built up from a set of components such as an EER relationship type which is composed of at least two participators and any number of attributes. A model element belongs to a class of modeling constructs if it matches the given pattern. All metamodels are mapped to a very small set of modeling constructs. In contrast, we regard the differences in the semantics of modeling constructs in different metamodels as subtle but important. For example, modeling sets with object identity and sets without object identity in the same way results in hiding this knowledge in code of the model management system whereas it should be part of the generic representation. In our representation we describe a model element by the set of roles it plays and their relationships to other elements. A small difference between two constructs can be modeled by adding a role to an element and thereby adding a new feature to the element.

The schema translations described in [2] are similar to ours in that they are also based on rules. One of the main differences to our approach is that MIDST specifies all facts about one model element in one term, whereas our terms about model elements use reification, the benefits of which we will explain in section 3.

QVT (Queries, Views, Transformations, `http://www.omg.org/docs/ad/02-04-10.pdf`) is a standard for model transformation in the context of MDA (model driven architecture). It comprises declarative and imperative languages. However, there does not yet exist a complete implementation of this standard.

Another important implementation of model transformation which allows declarative and procedural specification of transformations is the Atlas Transformation Language (ATL) [10, 11]. Like QVT, ATL deals with transformation in the context of MDA, and employs a hybrid transformation approach, i.e. the transformation is specified using *declarative* rules and *imperative* code. Rules and helper functions can be defined using a version of OCL (object constraint language) that has been extended to allow the handling of multiple metamodels. In QVT and ATL the rules that transform models are always defined on the metaclasses of the respective metamodels. That is, they are aware of the respective modeling languages. On the other hand, our goal is to allow *generic* model transformations that are indifferent to the employed metamodel. Therefore, the QVT and ATL approach is not appropriate for us.

2 Challenges and Requirements

A generic metamodel allows polymorphic usage of operators. However, in this setting if operators apply to models represented in the generic metamodel, models represented in native metamodels must be transformed into the generic metamodel and vice versa. The implementation of Import and Export operators can also be used as a validation of *GeRoMe* by proving that the modeling constructs from various metamodels can be imported and exported without loss of information. While implementing the mapping rules for the import and export operators, we have to assert that structures or constraints are uniquely imported into our metamodel and, vice versa, that *GeRoMe* represents these features non-ambigously, so that they can be exported again into the native format.

However, a formal proof of the correctness of such transformations is not possible. In [2] the authors already argued for their system that a formal proof of losslessness of translations to a generic metamodel is hopeless as even a test for losslessness of translations between two native metamodels is undecidable [4]. However, we tried to ease the formulation of such a mapping by implementing import and export in a way which allows the developer to concentrate on defining the mapping rules in a declarative way rather than distributing the mapping over a set of Java classes.

This section lists the requirements which this import and export functionality must meet.

1. Human-readable correspondences. Specifications of correspondences of concrete metamodel elements with their respective *GeRoMe* representations should be easily human readable in order to increase maintainability and extensibility.

2. Set-at-a-time translation. Usually, the transformation of models has to consider several model elements at a time. Therefore, Import and Export operators should work on a set-at-a-time basis rather than using complex navigational code which can only address one object at a time.

3. Mutually consistent Import and Export. Both translation operators for one native metamodel must rely on the same specification of inter-metamodel correspondences in order to prevent inconsistencies. Thus, correspondences specify equivalence of representations.

4. Define correspondences in a declarative way. Knowledge about correspondences of modeling constructs must not be hidden in imperative code of the system. Otherwise, the last requirement would be violated since different implementations had to be used for

import and export. Declarative translation rules are more transparent than imperative code.

5. Incremental development of translations shall be possible. Because relationships between the modeling constructs may be very complex, it must be possible to formalize them independently from each other without causing undesirable side effects. Thus, we have to ease an incremental implementation of translation operators.

6. Generation of elements shall be possible. Translation of models between metamodels requires new model elements to be generated. For example, as stated in [12] XML Schema elements are modeled in *GeRoMe* as associations with two association ends. In XML Schema, the association ends are given implicitly by the nesting structure of the element declarations. Thus, during the import into *GeRoMe* new model elements have to be generated. Hence, it must be possible to generate identifiers for these new model elements. They must be reproducible as the same generated model elements may occur in the consequents of multiple rules.

3 Implementation

Applications dealing with models require support for model management in several ways. Our goal is to provide a library for the management of models (including the definition of several operators) that can be reused. Our implementation of a model management platform contains an object model of our metamodel and a set of model management operators working on models represented in *GeRoMe*. Operators are not aware of the original metamodel, i.e. their implementations use only roles and structures in *GeRoMe*.

The requirements given in the preceding section strongly suggest to use a rule based approach for the implementation of Import and Export operators in which the antecedent of a rule selects the set of model elements and the consequent describes the resulting elements in the target metamodel. Since both, the Import and the Export operator for a modeling language, must rely on the same set of rules, our rules must be given as statements of equivalence between representations. This makes them applicable in both directions.

The equivalence rules are evaluated by a meta-program implemented in Prolog, which is able to handle rules with multiple predicates on both sides. Depending on whether the rules are used for import or export, the rule set is evaluated from left to right or the other way around. Our implementation uses SWI Prolog (`http://www.swi-prolog.org`) as underlying Prolog engine. In addition, the JPL package (Java Prolog Library) is used as a bridge between Java and Prolog. We currently use the Import and Export operators in an editor component for models represented in *GeRoMe*, in a matching system for schemas represented in different modeling languages, and in an implementation of the Merge operator that produces an integrated schema from two input schemas and a mapping in between.

Our logical notation of models uses reification to specify terms about *GeRoMe* model elements. The fact `modelElement(ID)` states that an object `ID` is a model element. Role objects are not explicitly represented; they are denoted as terms which have the name of their role class as functor and all objects on which they depend as arguments. For example, `attribute(ID)` states that the model element `ID` plays the *Attribute* role. The same term can be used to identify the role object. The $attr$ relationship is also reified: a term

like `attrName(o, v)` denotes that the object `o` has the value `v` for the attribute called `attrName`. For example, `max(attribute(ID), 1)` specifies that the *max*-attribute of the role object defined above is 1.

3.1 Import and Export

Fig. 2 shows an example. The rules are expressed in standard Prolog syntax, i.e. labels starting with an upper-case letter denote variables. The evaluation of the rules by Prolog and their definition as equivalence rules satisfy requirements 2 (set-at-a-time translation) and 3 (mutually consistent Import and Export).

The example defines the import of a column of a SQL table into a *GeRoMe* model. The column with the identifier `ID` belongs to a table and has a name and a type. In *GeRoMe*, we will create a model element with the same `ID`. The second statement defines the relationship between the namespace role of the table and the newly created model element. The following statements define that the element is visible and has a name. Then, we have to specify that the new model element plays also the *Attribute* role, and link this role to the *Aggregate* role of the model element representing the table. Finally, the type of the attribute is defined by linking it to the *Domain* role of the type, and the maximum cardinality of the attribute is set to 1.

With reification each model element in a *GeRoMe* model is described by a set of facts stating that the model element exists, which roles the element plays and which attribute values its roles have. Besides improving the readability (requirement 1) reificiation has more advantages. Often the translation of a model element only differs slightly when one of its properties differs. For instance, a SQL column will always be represented as a *GeRoMe* attribute. But if it is nullable, then the attribute will have a minimum cardinality of 0, otherwise it will be 1. If we would not use reification, but had used one large term of the form `sql_column(ID,TableID,Name,Type,Nullable)` we had to specify two almost identical rules for the translation between columns and attributes. These two rules would differ only in the two cases of `Nullable` and in the resulting values for the `max` attribute. By using reification, we can split the two large identical rules into three small rules, two of which deal with the cases of cardinality, and the other translates the remaining properties of the model element. Thus, reification allows reduction of code copy which is in general a good practice.

Another important term in this rule is `:translateType(Type, GeRoMeType)`. It translates SQL domains into their *GeRoMe* representations and vice versa. A variable must be used for the attribute type in the rule, otherwise the rule has to be copied for each possible type. The `translateType` predicate is a mapping of types between the two metamodels. This translation is necessary in both directions (import and export), and consequently, the same term must be included on both sides of the rule.

The leading colon is a syntactic convention which indicates that this term should be handled like a normal Prolog predicate and is ignored if it occurs in the consequent of a rule. Sometimes it is necessary to check more complex conditions before an element can be

```
sql_column(ID),
sql_column_table(ID, TableID),
sql_column_name(ID, Name),
sql_column_type(ID, Type)
:translateType(Type, GeRoMeType) <=>
  modelElement(ID),
  owned(namespace(TableID), ID),
  visible(ID),
  name(visible(ID), Name),
  attribute(ID),
  property(aggregate(TableID), attribute(ID)),
  type(attribute(ID), domain(GeRoMeType)),
  max(attribute(ID), 1),
  :translateType(Type, GeRoMeType).

sql_column_nullable(ID, true) <=>
   min(attribute(ID), 0).

sql_column_nullable(ID, false) <=>
   min(attribute(ID), 1).
```

Figure 2: Example rules for the Import/Export of SQL models

imported which can be implemented by custom predicates.

Some predicates in fig. 2 (e.g. `owned(..)`) have terms as arguments. As described above, these terms represent the role objects. From a logical perspective, we can interpret them as *Skolem functions*, which have been introduced on the right hand side to replace existentially quantified variables. As the goal is to construct objects using the *GeRoMe*-API, these functions must return meaningful objects. Therefore, when we create the *GeRoMe* objects from a set of facts, these functions will return the corresponding role objects of the given model elements, e.g. `attribute(ID)` returns the *Attribute* role of the model element `ID`. In doing so, we make sure that the same objects are used if they are referenced in different rules; for example, the attribute role of `ID` is referenced in all rules of fig. 2.

A built-in skolem function is `generateId(..)` that can be used to generate a unique identifier for a new model element from the arguments. If the generated identifier is needed in more than one rule to enrich it with more features, we can reproduce the identifier from the arguments. In doing so, requirement 6 is satisfied. Additionally, `generateId` sorts the identifier components lexically to ensure that the generated identifier does not depend on the order of the input terms.

The same rules are used for importing and exporting models. Depending on the target metamodel, a ruleset is activated and evaluated based on a set of facts representing the *GeRoMe* model. The given rule translates all model elements satisfying the right hand side of the rule to SQL columns.

Furthermore, our Export operators act credulously in that they do not check any constraints on the *GeRoMe* model which must be satisfied for it to be able to be exported to the target modeling language. Thus, prior to exporting a model to a native metamodel, problematic constructs (i.e. constructs which are not allowed in the target metamodel) must be transformed by ModelGen operators.

It must be emphasized that an import to and an export from *GeRoMe* might result in a

model different from the original model, as there are redundant ways to represent the same modeling construct in specific metamodels. For example, consider an OWL object property described as being functional; this could also be modeled by an *inverseFunctional* statement of the inverse property. In the import/export rules, such ambiguity will be resolved by using negation, e.g. the property will be defined as functional only if there is no (visible) inverse property that could be declared as *inverseFunctional* or vice versa.

3.2 Using Reflection

We do not plan to develop only a system for model transformation but a toolbox for model management in general. Thus, although we deem ModelGen to be best developed declaratively, other model management operators such as Match are better implemented in an object oriented language. Therefore, we must be able to traverse from declarative representations of models to object oriented representations. We used reflection to implement this transition for both, import and export. When importing, an `AbstractImport` operator translates the generated facts about the *GeRoMe* model to calls to the API which create an object model of the schema. When exporting, the Prolog facts about a *GeRoMe* model are created from the object model using reflection as well.

When a new native metamodel is to be supported we only have to implement two classes. One is, for instance, `ImportXSD` that traverses the model to be imported (e.g. using an XML Schema API), and generates facts in the representation to be used by the rules. Although this is not a strong requirement, the elements in the native modeling language should be described using reification because this will simplify the definition of the rules, as discussed in the examples above.

The Export operator works analogously. It uses the same rule file as the import of models, but interprets the rules from right to left. The `ExportXSD` class implements a hook method that creates the native model from the derived facts, again, using an appropriate API.

Due to the role and rule based approach and the generic implementation of the necessary Java classes the effort of supporting a new metamodel is minimized. Requirement 4 (define correspondences declaratively) is satisfied as the two classes to be implemented merely produce (or read) a special syntactic representation of the model and do not perform any sophisticated processing of the models. Since correspondences are not hidden in imperative code but given as a set of equivalence rules, the developer can concentrate on the logical correspondences and does not have to deal with implementation details. For example, import and export of SQL requires about 250 lines of Java code for each operator, and about 200 lines of code for the Prolog rules. The relationships between the modeling constructs could be expressed in less than 20 equivalence rules. Due to our approach the developer does not need to take all modeling constructs into account in the first place. Facts that are not used in rules are simply ignored. Since reification is used, the developer need not even consider all features of a single model element but can incrementally add detail to the rules. Thus, requirement 5 (incremental development of translations) is met. On the other hand if the role and rule based approach had not been used, a developer could easily get lost in lengthy translation classes.

4 Generic ModelGen Requirements

In model management, transformations of models between different metamodels, such as the classical EER to relational mapping or the derivation of a relational model from an XML Schema or vice versa are performed by so called ModelGen operators. These operators mainly focus on transforming modeling constructs allowed in the source schema but disallowed in the target schema to modeling constructs allowed in the target schema [2].

Similarly to Import and Export operators, ModelGen performs set-at-a-time translations on the model elements in a schema based on some precondition. Therefore, we aim at implementing these operators based on the same approach that we used for Import and Export operators. There are also commonalities in the implementation requirements of these operators:

- The semantics of ModelGen operators should not be hidden in procedural code but should be implemented in a way which improves readability, maintainability, extensibility, and incremental development.
- The effort of developing new ModelGen operators should be minimized. Using the rule based approach, the implementation effort is restricted to specifying the necessary translation rules. The existing classes for generating facts about models and creating objects from facts will only have to be adapted once to reflect possible extensions of our rules.
- ModelGen operators require new model elements to be generated. Consequently, it must be possible to generate identifiers for these new elements. Thus, we can use the same solution as for the import/export operators.

However, there are also new requirements for ModelGen:

1. Allow user interaction. Whereas Import and Export operators must and can be fully automated, ModelGen often requires decisions to be made where the required knowledge is not encoded in the input model. For instance, removing *IsA* relationships from an EER model requires knowledge about expected queries to the target model. Consequently, an appropriate hook for user interaction must be included into the rule interpreter.

2. Unidirectional rules. As ModelGen operators often delete model elements or role objects disallowed in the target model, they almost always reduce the information encoded in the model. Thus, their rules will usually be only applicable in one direction, unless the deleted information is re-entered by the user.

3. Traceability. Traceability of actions performed by ModelGen is required, such that a user can recognize which elements have been generated, deleted, or changed. Otherwise, it will be impossible to verify the result.

4. Mapping generation. During model transformation a mapping from the original model to the resulting model should be iteratively developed. That is, it must be updated each time a modeling construct is transformed. This also implies that after transformation the original model still exists and the resulting model is a new version of the original model. We believe that this can be reflected by introducing time stamps or state counters to the

```
[ modelElement(AS), modelElement(SourceAE), modelElement(TargetAE) ]
[ association(AS),
  objectAssociationEnd(SourceAE),
  objectAssociationEnd(TargetAE),
  property(association(AS), objectAssociationEnd(SourceAE)),
  property(association(AS), objectAssociationEnd(TargetAE)),
  participator(objectAssociationEnd(SourceAE), SourceType),
  participator(objectAssociationEnd(TargetAE), TargetType),
  max(objectAssociationEnd(SourceAE), 1),
  max(objectAssociationEnd(TargetAE), N), N > 1,
  :arity(association(AS), 2),
  name(visible(KeyAtt), KeyAttName),
  property(aggregate(TargetType), KeyAtt),
  type(attribute(KeyAtt), Domain),
  component(injective(ID), attribute(KeyAtt)),
  identified(identifier(injective(ID)), aggregate(TargetType)),
  :generateID(referenceTo(KeyAtt), RefAtt),
  :generateID(fk(TargetAE), FK)
] => [
  modelElement(RefAtt), attribute(RefAtt),
  type(attribute(RefAtt), Domain),
  component(FK, reference(attribute(RefAtt))),
  aggregate(SourceType), property(aggregate(SourceType),
  attribute(RefAtt)), reference(attribute(RefAtt)),
  referenced(attribute(RefAtt), attribute(KeyAtt))
  visible(RefAtt), name(visible(RefAtt), KeyAttName) ]
```

Figure 3: Example rule transforming object references to attribute references

model elements which are increased by application of the rules. Such counters will also ease tracing ModelGen as all intermediate states are preserved throughout subsequent transformation steps. One can collect all information about performed steps by querying Prolog for subsequent states.

5. Reusability. There are some transformation tasks which are common for several model transformation scenarios, such as transforming an EER model to a relational schema or transforming a UML model to a relational schema. In both cases, IsA relationships and complex attributes have to be transformed into corresponding elements in the relational schema. Thus, such "simple" transformations should be implemented as elementary ModelGen operators which can be reused in the definition of a complex ModelGen operator transforming a complete model.

Despite these differences we believe that our rule based implementation must only be slightly extended to meet the requirements of ModelGen.

Figure 3 shows an example of how a transformation rule could look like that satisfies some of the requirements. The example depicts a rule for transforming object references to simple typed references (i.e. associations to foreign keys). It could be used for the EER-to-relational mapping but also for transforming XML schema elements to referential constraints which would satisfy the reusability requirement.

The rule is partitioned into three parts. The first two parts are the antecedent of the rule, whereas the last part is the consequent. The terms in the rule do not contain state counters that distinct the different versions of the models. It is not necessary to include these counters explicitly as this can be easily handled by the interpreter just by incrementing the counter

of the elements in the consequent.

We assume that each transformation will usually transform only a small number of model elements, whereas the majority of elements can be copied by default from one version of the model to the next version. Therefore, we partition the antacedent into two parts. The first part contains the model elements that are transformed. Conditions on other elements, such as the `SourceType` which are to be copied to the target model, must occur only in the second part. Any element that occurs in the first part of a rule will not be copied to the target model. This works recursively; in the example, the model element `AS` representing the association and its association ends (`SourceAE` and `TargetAE`) and all of their roles will not be copied. Of course, references to any of the roles of deleted elements must also not be copied to the target model. The second part in the example specifies the conditions for the association: in this case, we are considering binary (arity of 2) one-to-many (the max cardinality of the association ends is 1 and N) associations. The remaining predicates in the second part retrieve the key attribute of the target aggregate. The two `generateID` predicates generate the identifiers for the model elements representing the reference (i.e. the foreign key constraint) and the attribute which has to be added to the source type. The corresponding model elements and their roles which should be present in the target model are then specified in the third part.

5 Lessons Learned and Outlook

The implementation of Import and Export operators in model management underlies certain requirements, such as mutally consistent import and export, declarative definition, and maintainability. A role and rule based approach is a perfect fit for realizing these requirements among others. Furthermore, the usage of reification drastically reduces the lines of code of translation rules by avoiding copying of large portions of translation rules.

Using a rule-based approach for specifying the Import and Export operators has the advantage that the semantics of these operators can be specified in a declarative way and is not hidden in the code of a complex transformation function. In addition, our approach is fully generic; it uses reflection and annotations in Java to create objects or to generate facts from an existing *GeRoMe* model. Therefore, the code required to support another metamodel is limited to the generation of the metamodel-specific facts and the specification of the equivalence rules. This reduces the effort for the implementation of the operators significantly.

In the near future we will extend our implementation in order to realize rule based ModelGen operators. These operators will perform metamodel independent transformations of models represented in our generic metamodel. Certain enhancements are necessary for this task, such as the introduction of state counters, traceability of actions, and hooks for user interaction. However, the current implementation of Import and Export seems to be an adequate basis for an implementation of rule based set-at-a-time ModelGen operators.

References

[1] P. Atzeni, P. Cappellari, and P. A. Bernstein. A multilevel dictionary for model management. In *Proc. Conf. Conceptual Modeling (ER)*, volume 3716 of *LNCS*, pages 160–175. Springer, 2005.

[2] P. Atzeni, P. Cappellari, and P. A. Bernstein. Model-independent schema and data translation. In *EDBT*, volume 3896 of *LNCS*, pages 368–385. Springer, 2006.

[3] P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *Proc. EDBT*, volume 1057 of *LNCS*, pages 79–95. Springer, 1996.

[4] D. Barbosa, J. Freire, and A. O. Mendelzon. Information preservation in XML-to-relational mappings. In *Proc. XSym 2004*, volume 3186 of *LNCS*, pages 66–81. Springer, August 2004.

[5] P. A. Bernstein. Applying model management to classical meta data problems. In *Proc. CIDR'03*, Asilomar, CA, 2003.

[6] P. A. Bernstein, A. Y. Halevy, and R. Pottinger. A vision for management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[7] P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength schema matching. *SIGMOD Record*, 33(4):38–43, 2004.

[8] M. A. Hernández, R. J. Miller, and L. M. Haas. Clio: A semi-automatic tool for schema mapping. In *Proc. ACM SIGMOD'01*, page 607, Santa Barbara, CA, 2001. ACM Press.

[9] ISO/IEC. Information technology – information resource dictionary system (IRDS) framework. International Standard ISO/IEC 10027:1990, DIN Deutsches Institut für Normung, e.V., 1990.

[10] F. Jouault and I. Kurtev. Transforming models with ATL. In *MoDELS 2005 Revised Selected Papers*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.

[11] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195. ACM Press, 2006.

[12] D. Kensche, C. Quix, M. A. Chatti, and M. Jarke. *GeRoMe*: A generic role based metamodel for model management. *Journal on Data Semantics*, VIII:82–117, 2007.

[13] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *Proc. SIGMOD*, pages 193–204. ACM, 2003.

[14] R. Pottinger and P. A. Bernstein. Merging models based on given correspondences. In *Proc. VLDB*, pages 826–873. Morgan Kaufmann, 2003.

[15] C. Quix, D. Kensche, and X. Li. Generic schema merging. In *Proc. 19th Intl. Conference on Advanced Information Systems Engineering (CAiSE'07)*, Trondheim, Norway, 2007. to appear.

[16] G.-L. Song, K. Zhang, and J. Kong. Model management through graph transformation. In *VL/HCC*, pages 75–82. IEEE Computer Society, 2004.