GeRoMe: A Generic Role Based Metamodel for Model Management

David Kensche[1], Christoph Quix[1], Mohamed Amine Chatti[1], Matthias Jarke[1,2]

[1]RWTH Aachen University, Informatik V (Information Systems), 52056 Aachen, Germany
[2]Fraunhofer FIT, Schloss Birlinghoven, 53574 St. Augustin, Germany
{kensche,quix,chatti,jarke}@cs.rwth-aachen.de

Abstract. The goal of *Model Management* is the development of new technologies and mechanisms to support the integration, evolution and matching of data models at the conceptual and logical design level. Such tasks are to be performed by means of a set of model management *operators* which work on models and their elements, without being restricted to a particular metamodel (e.g. the relational or UML metamodel).
We propose that generic model management should employ a generic metamodel (GMM) which serves as an abstraction of particular metamodels and preserves as much of the original features of modeling constructs as possible. A naive generalization of the elements of concrete metamodels in generic metaclasses would lose some of the specific features of the metamodels, or yield a prohibitive number of metaclasses in the GMM. To avoid these problems, we propose the *Generic Role based Metamodel GeRoMe* in which each model element is *decorated* with a set of role objects that represent specific properties of the model element. Roles may be added to or removed from elements at any time, which enables a very flexible and dynamic yet accurate definition of models.
Roles expose to operators different views on the same model element. Thus, operators concentrate on features which affect their functionality but may remain agnostic about other features. Consequently, these operators can use polymorphism and have to be implemented only once using *GeRoMe*, and not for each specific metamodel. We verified our results by implementing *GeRoMe* and a selection of model management operators using our metadata system ConceptBase.

1 Introduction

Design and maintenance of information systems require the management of complex models. Research in (data) *model management* aims at developing technologies and mechanisms to support the integration, merging, evolution, and matching of data models at the conceptual and logical design level. These problems have been addressed for specific modeling languages for a long time. Model management has become an active research area recently, as researchers now address the problem of *generic* model management, i.e. supporting the aforementioned tasks without being restricted to a particular modeling language [7,8]. 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.

According to the IRDS standard [18], metamodels are *languages* to define models. Examples for metamodels are *XML Schema* or the *UML Metamodel*. 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. 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.

1.1 The Challenge: A Generic Mechanism for Representing Models

This paper addresses the first challenge mentioned in [8], the development of a mechanism for representing models. Since the goal is the support of *generic* model management, this has to be done in some generic way. Currently, model management applications often use a generic graph representation but operators have to be aware of the employed metamodel [10,15,23]. A graph representation is often sufficient for the purpose of finding correspondences between schemas, which is the task performed by the model management operator Match [28], but such a representation 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 [27] a generic (but yet simple) metamodel is used that distinguishes between different types of associations in order to merge two models. Consequently, in order to support a holistic model management framework it is necessary to provide a detailed generic metamodel. A more detailed discussion about the related work on the representation of models is given in section 2.

The intuitive approach to develop a truly generic metamodel (GMM) identifies abstractions of the metaclasses of different metamodels. Its goal is to define a comprehensive set of generic metaclasses organized in an inheritance lattice. Each metaclass in a given concrete metamodel then has to be mapped to a unique metaclass of the GMM.

The sketched approach exhibits a prohibitive weak point: elements of particular metamodels often have semantics that overlap but is neither completely different nor equivalent. For example, a generic Merge operator has to merge elements such as classes, relations, entity types and relationship types. All of these model elements can have attributes and should therefore be processed by the same implementation of an operator. In this setting, such polymorphism is only possible if the given model elements are represented by instances of the same metaclass in the GMM, or at least by instances of metaclasses with a common superclass. Thus, one has to choose the features of model elements which are combined in one metaclass.

Actually, in each metamodel there may be elements incorporating an entirely new combination of such aspects. One approach to cope with this problem is to focus on the "most important" features of model elements while omitting such properties which are regarded as less important. But to decide which properties are important and which are not results in loss of information about the model.

All properties of model elements could be retained if the GMM introduced a set of metaclasses as comprehensive as possible and combined them with multiple inheritance such that any combination of features is represented by a distinct metaclass. Despite the modeling accuracy of such a GMM, it will suffer from another drawback, namely that it leads to a combinatorial explosion in the number of sparsely populated intersection classes which add no new state.

1.2 Our Solution: Role Based Modeling

In such cases, a role based modeling approach is much more promising. 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 GMM, 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 or evolution, an element may gain or lose roles, thereby adding and revoking features. Thus, the combinatorial explosion in the number of metaclasses is avoided but nevertheless most accurate metadata modeling is possible.

Therefore, the GMM proposed in this work retains these characteristics by employing the *role based* modeling approach, resulting in the *Generic Role based Metamodel GeRoMe* (phonetic transcription: dʒerəʊm). Implementations of model management operators can assert that model elements have certain properties by checking whether they play the necessary roles. At the same time the operator remains agnostic about any roles which do not affect its functionality. Thus, while role based metamodeling allows to formulate accurate models, the models appear to operators only as complex as necessary. *GeRoMe* will be used only by model management applications; users will use their favorite modeling language.

The difference between our and the naive generalization approach is similar to the difference between the local-as-view (LAV) and global-as-view (GAV) approaches in data integration. By defining elements of a GMM as generalization of elements of specific metamodels, an element of the GMM is defined as a view on the specific elements. In contrast, in our approach the definition of the roles in *GeRoMe* is independent of a particular metamodel, and the elements of the concrete metamodels can be characterized as a combination of roles. Thus, our role based approach can be seen as a LAV approach on the meta level, which has similar advantages as the normal LAV approach [22]. The role based metamodel is more "stable" with respect to the concrete metamodels represented, i.e. additional modeling features of other metamodels can be easily added by defining new role classes. Thus, this change would not affect other role classes in *GeRoMe*. In addition, the representations of the concrete metamodels are more accurate as their elements can be described by a combination of role classes.

The definition of the GMM requires a careful analysis and comparison of existing metamodels. Since it has to be possible to represent schemata in various metamodels in order to allow generic model management, we analyzed five popular yet quite different metamodels (Relational, EER, UML, OWL DL, and XML Schema). We identified the common structures, properties, and constraint mechanisms of these metamodels. This part of our work can be seen as an update to the work in [17], in which several semantic database modeling languages have been compared.

The paper is structured as follows. Section 2 provides some background information on model management and role based modeling, and presents a motivating scenario. In section 3, we analyze and compare existing metamodels and derive the *Generic Role based Metamodel GeRoMe*. Section 4 shows several examples of models in different metamodels represented in *GeRoMe*. Section 5 explains how model management operations can be performed using *GeRoMe*. As an example, we describe some atomic operations necessary for the transformation of an EER model into a a relational model. The architecture and implementation of our model management prototype is discussed in section 6. In particular, we present a rule-based approach to import and export models. Finally, section 7 summarizes our work and points out future work.

2 Background and Motivation

The next subsection provides an overview of model management in general. The motivating scenario in section 2.2 should give an idea of the benefits of a model management framework and the usage of a generic metamodel for model management. An overview of work about role based modeling concludes this section.

2.1 Model Management

Model management aims at providing a formalization for the definition and modification of complex models [8]. To achieve this goal, a model management system has to provide definitions for *models* (i.e. schemas represented in some metamodel), *mappings* (i.e. relationships between different models), and *operators* (i.e. operations that manipulate models and mappings). There have been earlier approaches to model management [3,20], which did address especially the transformation of models between different metamodels. Model management has become more important recently, as the integration of information systems requires the management of complex models. The most important operations in model management are Merge (integration of two models), Match (creating a mapping between two models), Diff (finding the differences between two models), and ModelGen (generating a model from another model in a different metamodel representation).

Rondo [23] is the first complete prototype of model management. It represents models as directed labeled graphs. Each node of such a graph denotes one model element, e.g. an XML Schema complex type or relational table. A model is represented by a set of edges between these nodes. A model element's type (Table, Column, Class, ...) is also specified by such an edge with the label *type*. Furthermore, types of attributes are specified by other dedicated edges, e.g. *SQLtype*. For each of the supported metamodels a different set of types is available. Although the models are represented in a generic graph structure, the implementation of the operators is not truly generic. For example, the implementation of the Match operator requires two models of the same type as input, and some operators (such as Extract) have specific implementations for each metamodel.

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. They map all metamodels to a very small set of modeling constructs. In constrast, 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.

Another rule-based approach to model transformation is presented in [9]. Models are first translated into a universal metamodel and then a sequence of rule-based transformations is applied to generate a model that is valid in the target metamodel. Details about the universal metamodel are not given in [9].

Clio [15] is a tool for creating schema mappings. Whereas schema matching algorithms just discover correspondences between schemas, Clio goes one step further and derives a mapping from a set of correspondences. The mapping is a query that transforms the data from one schema into another schema. However, Clio supports only XML and relational schemas.

More sophisticated model management operators such as Merge (integration of two models according to a given mapping, resulting in a new model) require even more semantic information about the models involved. For example, in [27] a meta model with several association types (e.g. has-a, is-a) is used.

The various approaches to model management show that each operator requires a different view on a model. Schema matching focuses on labels and structure of schema elements, whereas merging and transformation of models require more detailed information about the semantics of a model (e.g. association types, constraints). These different views are supported by our role based approach, as operators will see only those roles which are relevant in their context.

2.2 Scenario

The following simplified scenario should provide an idea of what model management is about and of the benefits of utilizing a generic metamodel for model management.

Complex information systems undergo regular changes due to changes of the requirements, of the real world represented by the information system, or of other systems connected to the information system. As an example, we consider the following eBusiness scenario: a supplier of an automotive manufacturer receives orders from a business partner in some XML format (XS1). The orders are entered into the ERP system of the supplier by a transformation program, which uses a mapping between the XML schema and the relational DB (RM2) of the ERP system.

In order to generate this mapping, the two models are represented as models in a generic metamodel (GM1 and GM2). A Match operator can then be used to create a mapping GM1_GM2 between the two models, which can be further translated into the desired mapping XS1_RM2 between the original models, e.g. by exporting the mapping with an operator which generates a set of data access objects for RM2 and a parser

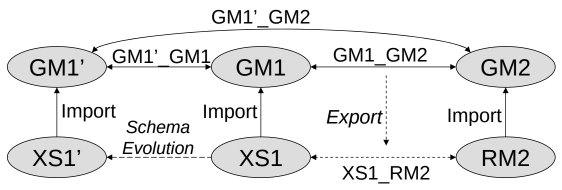

Fig. 1. Schema evolution using *GeRoMe* and Model Management

for XML documents conforming to XS1 as well as glue code which uses the mapping information for adapting these classes to each other.

Due to a change in the system of the manufacturer, the schema of the orders has changed. This change has to be propagated to the mapping between the XML schema and the relational DB. Focusing on the models, this scenario can be seen as an example of schema evolution (Fig. 1). The original XML schema XS1 is mapped to the relational model (RM2) of the DB using the mapping XS1_RM2. The schema evolution generates a new version of the XML schema, namely XS1'.

Again, instead of applying the model management operators to the level of specific schemas, we will first generate a corresponding representation of the specific model in *GeRoMe* (GM1'). Then, we have to apply the Match operator to GM1 and GM1', resulting in a mapping GM1'_GM1 between these models. This match operation should be simpler than matching the new version GM1' with GM2 directly, as two versions of the same model should be quite similar. Then, we can compose the mappings GM1'_GM1 and GM1_GM2 to a new mapping GM1'_GM2. Note, that this operation has just to consider mappings between models represented in *GeRoMe*, which should simplify the implementation of such an operator. The result of this step is a mapping from GM1' to GM2 of those elements which are also present in GM1.

In order to map elements which have been added during the schema evolution a Diff operator has to be used on GM1' and GM1 which takes into account the mapping GM1'_GM1. The difference then has to be mapped individually.

The important difference to other approaches is that the operations in *GeRoMe* are truly generic, they do not have to take into account different representations of models. Therefore, the operators have to be implemented only once, namely for the *GeRoMe* representation. In the example the same match operator can be used in both cases, to match the two versions of the XML Schema and to match the XML Schema with the relational model.

2.3 Role Based Modeling

The concept of role (or aspect) based modeling has first been described in detail in the context of the network model [4] and later on in several works on object-oriented development and object-oriented databases [11,29,30].

Different formalizations have been proposed, which exhibit significant differences, but all have in common that a role extends the features of an existing object while being a view on the object and *not* an object in its own right. In [11] *multiple direct class membership* is considered as a solution to the problem of artificial intersection classes.

That is, instead of defining an intersection class, the combination of state and behavior is achieved by defining an object to be instance of several classes at the same time, which are not necessarily on the same specialization path.

In [29] the notion of aspects of objects is discussed. It is stated that at any given moment an entity may have many different types that are not necessarily related. Often this issue cannot be handled by multiple inheritance since this would lead to a large number of sparsely populated "intersection classes" which add no new state. This approach is different from multiple direct class membership in that each object can have multiple aspects of the same type, e.g. a person can at the same time be a student at more than one university while still being the same individual.

[6] presents an approach to avoid large class hierarchies in chemical engineering applications that is also based on aspects. Aspects divide a class into separately instantiatable partitions. Thus, aspects are a "part" of the object whereas roles are more "external" objects attached to another object, thereby providing different views on that object. A comparison of aspects and roles and issues concerning their implementation are discussed in [14].

Other approaches, such as the one considered in [30], treat the different features of an object as roles, which are themselves instances of so called *role classes* and have identity by state. This representation also allows model elements to play directly or implicitly more than one instance of the same role. In addition, [30] introduces the concept of *role player qualification* which means that not every object may play every role but that certain conditions have to hold.

3 The Generic Role based Metamodel *GeRoMe*

In this section, we will first explain the role model which we have employed to define *GeRoMe*. Based on our analysis of existing metamodels (section 3.2), we have derived the *Generic Role based Metamodel*, which is described in detail in section 3.3.

3.1 Description of the Role Model

GeRoMe employs the following role model. A model element is represented by an object which has no characteristics in its own right. Roles can be combined to describe a model element encompassing several properties. Thus, the model element is *decorated* with its features by letting it *play* roles. A role maintains its own identity and may be player of other roles itself. Because a model element without roles does not have any features, every model element has to play at least one role. Every role object has exactly one player. In our model, some role classes may be used more than once by an element, e.g. an *Attribute* may play the role of a *Reference* to more than one other *Attribute*. Thus, the complete representation of a model element and its roles forms a tree with the model element as its root.

We used three different relationships between role classes, namely *inheritance*, *play*, and *precondition*. The *play* relationship defines which objects may be player of certain roles. For example, an *Attribute* role may play itself the role of a reference. In addition, a role may be a precondition of another role. Thus, in order to be qualified to play a role of a certain class, the player must be already the player of another role of a certain other class. Except for namespaces, all links between model elements are modeled as links between roles played by the elements.

To tap the full power of role modeling, we have to define role classes in such a way that each of them represents an "atomic" property of a model element. Then roles can be combined to yield the most accurate representation of an element.

3.2 Role Based Analysis of Concrete Metamodels

A generic metamodel should be able to represent both the structures and constraints expressible in any metamodel. Thus, to define such a metamodel it is necessary to analyze and compare the elements of a set of metamodels. Our choice of metamodels comprises the relational model (RM) [12] and the enhanced entity relationship model (EERM) [12] because these two models are rather simple and are in widespread use. The metamodel of the Unified Modeling Language (UML, version 1.5) has been analyzed as an example for object-oriented languages. The description logics species of the Web Ontology Language (OWL DL, `http://www.w3.org/2004/OWL/`) has been included since it follows different description paradigms due to its purpose. For example, properties of concepts are not defined within the concepts themselves but separately. Finally, XML Schema (`http://www.w3.org/XML/Schema`) has been analyzed as it is the most important metamodel for semistructured data.

We analyzed the elements and constraints available in these five metamodels and identified their differences and similarities. In doing so, we determined the role classes, which constitute our role based metamodel. In total, we compared about seventy structural properties and elements and twenty types of constraints. Some of them are very easily abstracted, such as data types or aggregates. Others, such as the XML Schema *element* or OWL object properties, are rather intricate and need closer inspection. The XML Schema element is an association (associating a parent element with its children). The root element of a document is a special element which does not have a parent. Furthermore, an XML Schema may allow different types of root elements for a document. Another problematic example are object properties in OWL DL: the *Association* role is played by a "pair of properties" and the *ObjectAssociationEnd* role is played by object properties. Furthermore, some metamodels provide redundant options for representing the same semantics, e.g. there is no semantic difference between an XML Schema attribute and a simple-typed XML Schema element with a maximum cardinality of 1. Thus, it is difficult to represent such specific model elements in a GMM. In section 4, we describe some of the representation problems in more detail.

Table 1 shows a selection of role classes and states the related model elements in the considered metamodels. The table contains roles which are used to define structural model elements (e.g. relation, class) and roles to define relationships and constraints (e.g. association, disjointness). Due to space constraints, the table does not embody all metamodel elements and correspondences in the different metamodels.

3.3 Description of *GeRoMe*

Figure 2 presents the Generic Role based Metamodel *GeRoMe* at its current state, based on the analysis of the previous section. All role classes inherit from *RoleObject* but we omitted these links for the sake of readability. Although we use here the UML notation

Role	**EER**	**Relational**	**OWL DL**	**XML Schema**	**UML**
Domain	domain	domain	xsd datatype	any simple type	datatype
Aggregate	entity/rel.-ship type, comp. attr.	relation	class	complex type	class, association class, struct
Association	relationship type	-	a pair of inverse object properties	element	association, association class
ObjectSet	entity/rel.-ship type	-	class	complex type, schema	class, ass. class, association, interface
Base-Element	supertype in isA, subset in Union	base of anonymous domain	superclass, superproperty	base simple / complex type	superclass, implemented interface
Derived-Element	subtype in isA or union type	anonymous domain constraint	subclass, subproperty	derived simple / complex type	subclass, subinterface, implementation
Union	derivation link of union type	-	derivation link of union class	derivation link of union type	-
IsA	isA derivation link	-	subclassing derivation link	restriction / extension derivation link	subclassing, implementation
Enumeration	enumerated domain restriction	enumerated domain restriction	enumeration	enumeration	enum, constants in interface (constant inheritance)?
Attribute	(composite / multivalued) attribute	column	data type property	attribute, element with simple type	attributes in struct, member variables, properties
Object-Association-End	link between relationship type and its participator	-	object property	link between element and its nested or enclosing complex type	point where association meets participator
Literal-Association-End	-	-	-	link between an element and its nested simple type	-
Literal	instance of a domain	domain value	data type value	simple type value	constant, value of simple type
Structured-Instance	instance of a structured type	tuple	individual	valid XML	value of struct, object
Visible	entity type, rel.-ship type, attr.	relation, column	named class, property	named type, attribute element	anything not anonymous
Reference	-	foreign key comp.	-	keyref component	-
Foreign Key	-	foreign key	-	keyref	-
Disjointness	constraint on subtypes	-	constraint on classes	-	constraint on classes
Injective	primary/partial key	unique, primary key	inverse functional	unique, key	-
Identifier	primary/partial key	primary key	-	key	-
Universal	anonymous domain of attribute	anonymous domain constraint of column	allValuesFrom	restriction of complex type	- (covariance breaks polymorphism)
Existential	-	-	someValuesFrom	-	-
Default	-	default value	-	default value	default value

Table 1. Roles played by concrete metaclasses

to describe the *metamodel GeRoMe*, it has to be stressed that UML or the related MOF standard (`http://www.omg.org/mof/`) are not suitable for expressing *models* for *generic* model management applications, since – as we discussed above – the use of multiple inheritance instead of a role based approach would lead to a combinatorial explosion of classes in the metamodel. Below, we will describe the elements of *GeRoMe* according to their basic characteristics: structural elements, derivations, and constraints.

Structural Elements Every model element representing a primitive data type plays the role of a *Domain*. *GeRoMe* contains a collection of predefined domains such as *int* and *string*. In contrast, model elements which may have attributes play an *Aggregate* role (e.g. entity and relationship types, composite attributes in EER; relations, classes and structs in other metamodels).

Thus, the *Aggregate* role is connected to a set of *Attribute* roles. Each of these *Attribute* roles is part of another tree-structured model element description. An *Attribute* role is a special kind of *Particle* and has therefore the *min* and *max* attributes which can be used to define cardinality constraints. Every attribute has a *Type*, which may be a primitive type or an *Aggregate* in the case of composite attributes.

The *Aggregate* role and the *Domain* role are specializations of *Type*. *Type* is a specialization of *DerivableElement* which is the abstract class of roles to be played by all model elements which may be specialized. Another kind of *DerivableElement* is the *Association* role. Properties of associations are *AssociationEnd* roles. For example, association roles are played by EER relationship types, UML associations, or UML association classes. A model element which provides object identity to its instances may participate in one or more associations. This is modeled by specifying the element's *ObjectSet* role to be the participator of one or more *ObjectAssociationEnd* roles. Thus, an association end is a model element in its own right, and the association is a relationship between objects and values. In addition, the roles *AggregationEnd* and *CompositionEnd* can be used to model the special types of associations available in UML. In order to be able to represent the aforementioned special case of XML Schema elements having a simple type, we had to introduce the *LiteralAssociationEnd* as a role class. Furthermore, an *Attribute* or *LiteralAssociationEnd* role may itself play the role of a *Reference*, which defines a referential constraint referencing another *Attribute* of the same type.

The *Association* and *Aggregate* role classes are an intuitive example of two role classes that can be used in combination to represent similar concepts of different metamodels. If the represented schema is in a concrete metamodel which allows relationship types to have attributes, such as the EER metamodel, then every model element playing an *Association* role may play additionally an *Aggregate* role. If associations may not have attributes, which is the case in OWL, a model element may only play either of both roles. On the other hand, the representation of a relational schema may not contain *Association* roles at all. Thus, these two roles can be combined to represent the precise semantics of different metamodel elements. Of course any of these combinations can be further combined with other roles, such as the *ObjectSet* role, to yield even more description choices.

We have defined a formal semantics for models represented in *GeRoMe* that allows to specify *Instances* for model elements which play a *Set* role. Values of *Domains* are modeled as elements playing a *Literal* role. On the other hand values of elements play-

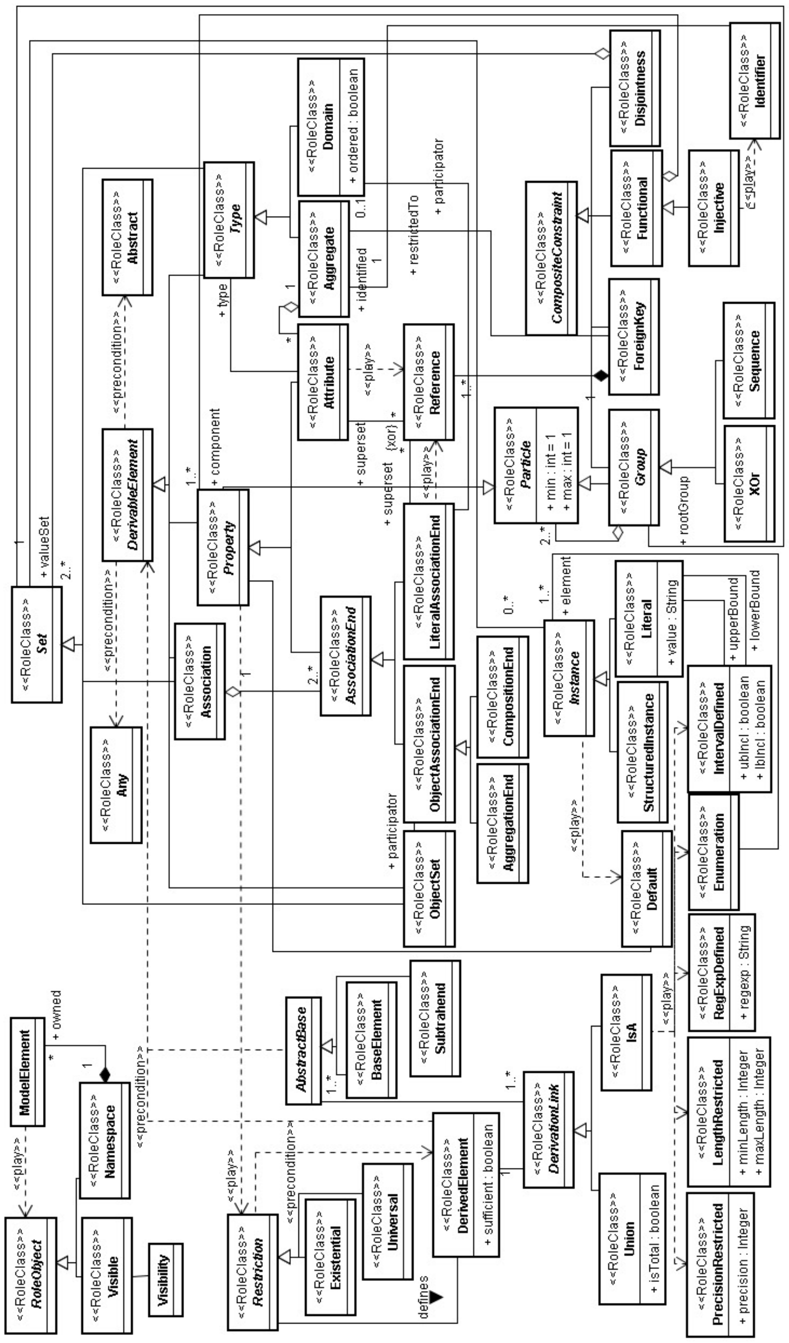

Fig. 2. The Generic Role based Metamodel (*GeRoMe*)

ing *ObjectSet*, *Aggregate*, or *Association* roles, or combinations thereof are represented by elements playing a *StructuredInstance* role. These are for example rows in a table, values of structs in UML, or instances of classes or association classes. An *Abstract* role marks a *Set* as being not instantiable. The *Any* role is used as a wildcard, in cases where types or associations or attributes are not constrained. This is commonly used in XML Schema where you can specify components of a complex type with `anyAttribute` or `anyElement`, for instance. Each *Instance* can also play the role of a *Default* value with respect to any number of properties. Our formal semantics defines also the shape of the structured instance such that it conforms to the structure defined by its value set. But we abstain here from further elaborating on that issue since this topic abandons the model level for the instance level.

Finally, model elements can be *Visible*, i.e. they can be identified by a name. The *name* attribute of a *Visible* role has to be unique within the *Namespace* it is defined in. Furthermore, a visibility can be chosen for a *Visible* element from a predefined enumeration. A model's root node is represented by a model element which plays a *Namespace* role.

Derivation of New Elements A *BaseElement* role is played by any model element that is a superset in the definition of a derived element. Thus, a *DerivedElement* can have more than one *BaseElement* and vice versa. These roles can be played by any *DerivableElement*.

The *BaseElement* and *DerivedElement* roles are connected via dedicated model elements representing the *DerivationLink*. Each *DerivationLink* connects one or more *BaseElements* to one *DerivedElement*. The *IsA* role can be used to define specialization relationships. It extends the definition of a superclass by adding new properties (e.g. inheritance in UML). A *DerivedElement* role which is connected to an *IsA* role with more than one *BaseElement* role can be used to define a type which is the intersection of its base elements. A *Subtrahend* is an element whose instances are never instances of the derived element (e.g. a `complementOf` definition in OWL).

We identified two different kinds of *isA* relationships which are often not distinguished from each other. All surveyed metamodels allow *extension* (i.e. the subtype defines additional attributes and associations) if they allow specialization at all. In EER and OWL, model elements can specialize base elements also by constraining the ranges of inherited properties. In EER, this is called *predicate defined specialization* [12, p.80], whereas in OWL it is called *restriction* and comprises a very important description facility for inheritance. Such derivations can be expressed in our metamodel by deriving the constrained property from the original one and letting it play the role of a *Universal* or *Existential* restriction. This *Restriction* role must reference the *DerivedElement* role of the respective subclass. These restrictions cannot be used in UML. For example defining a universal restriction on an association would amount to covariance, that is specialization of a property when specializing a class. Covariance breaks polymorphism in UML (or object oriented programming languages); it is therefore not allowed.

Special kinds of derivations are for example enumerations and intervals. We model such derivations by letting the *IsA* link play additional roles. This is similar to the facets of XML Schema simple types and allows to orthogonally specify conditions of the derived *Set*. Obviously some of these roles may only be applied when deriving *Domains*.

You can define new structural elements by using an *Enumeration* role and enumerating those *Instances* which are element of the new *Set*. Furthermore, derivations may define intervals of existing *Domains* or restrict the length and precision of their values. In case the base element is the built-in domain *string* or a subtype thereof a regular expression can define a new subtype. In XML Schema, named domains can be derived from others whereas in the relational metamodel derived domains occur only as an anonymous type of attributes with enumeration or interval domains.

Constraints Constraints are represented by separate model elements. For example, a disjointness constraint on a set of derived elements (or any other types) has to be defined by a model element representing this constraint. The element has to play a *Disjointness* role which references the types to be disjoint. In the case of OWL or UML, any collection of classes can be defined to be disjoint. When representing an EER model, this constraint can be used to define a disjoint *isA* relationship by referencing at least all of the derived elements.

Another constraint is the *Functional* constraint which declares a property or a set of properties to have the characteristics of be a function (uniqueness and completeness) and is used for example to represent certain OWL properties. Correspondingly, an *Injective* property is a functional property that specifies a one-to-one relationship. Such an *Injective* role is equivalent to a uniqueness constraint in XML Schema or SQL. It can also define a composite key by being connected to multiple properties. An injective constraint playing an *Identifier* role defines a primary key. This reflects the fact that a primary key is only a selected uniqueness constraint, and thus, only one of multiple candidate keys.

The *ForeignKey* constraint is a collection of *Reference* roles which defines a (possibly composite) reference to an *Identifier*. This is used to model foreign keys in the relational model or key references in XML Schema.

Additional restrictions on the structure of *Aggregates* or *Associations* can be given by *Group* constraints which reference a set of *Particles*. For instance, the *Sequence* constraint defines the order of appearance of properties. The *XOr* constraint is a modeling feature that is available in the UML metamodel or in XML Schema. It states that an object may participate only in one of the related associations or that only one of referenced attributes occurrs. Such *Group* constraints can also be nested which corresponds to the nesting of the respective model groups in XML Schema and allows to define them recursively together with cardinality constraints.

We are aware that there are subtle differences in the semantics of constraints for the various metamodels. However, these differences stem from the objectives of the respective modeling languages and apply only to the data level. In contrast, the goal of *GeRoMe* is to represent models and to provide a generic data structure for manipulating them. For instance, in a relational database a uniqueness constraint is checked whenever a row is inserted or updated whereas in an ontology such a constraint will only narrow the interpretation of the model such that individuals with the same value for the unique property are classified as being equal. On the model level the constraint is just a statement about the property.

Another issue are constraints that can be attached as an expression in some formal constraint language to the model (e.g. OCL constraints or SQL assertions). Such constraints cannot be represented in a generic way, as this would require a language that unifies all features of the various constraint languages. Thus, a generic constraint language would be difficult to interpret because of the complexity of the language or it would be undecidable whether a constraint can be satisfied or not. Currently, we are able to express constraints as first-order logic formulas (using predicates referring to the instance level as defined in appendix A.1) which certainly cannot cover all constraints (e.g., SQL assertions with aggregations or functions). Therefore, a translation of existing contraint languages into our language could be done only partially. The opposite way, however, will be possible, e.g. generating executable code from these constraints. This is especially important for mappings between different models as these mappings will be used to transform data from one model into another model.

GeRoMe can be extended with new role classes representing other features of constraints and structures while existing models and operators still remain correct.

4 Representation Examples

This section presents some example models based on a small airport database in [12, p.109] (see fig. 3). We represented EER, XML Schema and OWL DL models for this example. The model contains simple entity types composed of attributes as well as some advanced features, which are not supported by all metamodels (e.g. composite attributes, *isA* relationship).

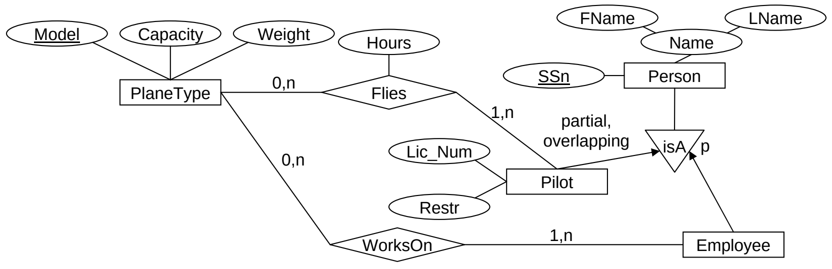

Fig. 3. Part of an airport EER schema

4.1 Representation of an EER Schema

Fig. 4 shows a part of the representation of the airport model in *GeRoMe*. The *GeRoMe* representation shows each model element as a *ModelElement* object (gray rectangle) which plays a number of roles (white squares) directly or by virtue of its roles playing roles themselves. Each such role object may be connected to other roles or literals, respectively. Thus, the roles act as interfaces or views of a model element. The links between role objects connect the model element descriptions according to the semantics of the represented schema.

For the sake of readability, we refrain here from showing the whole model and omitted repeating structures with the same semantics such as links from namespaces to their owned elements or *Visible* roles. A model element plays a *Visible* role if it has

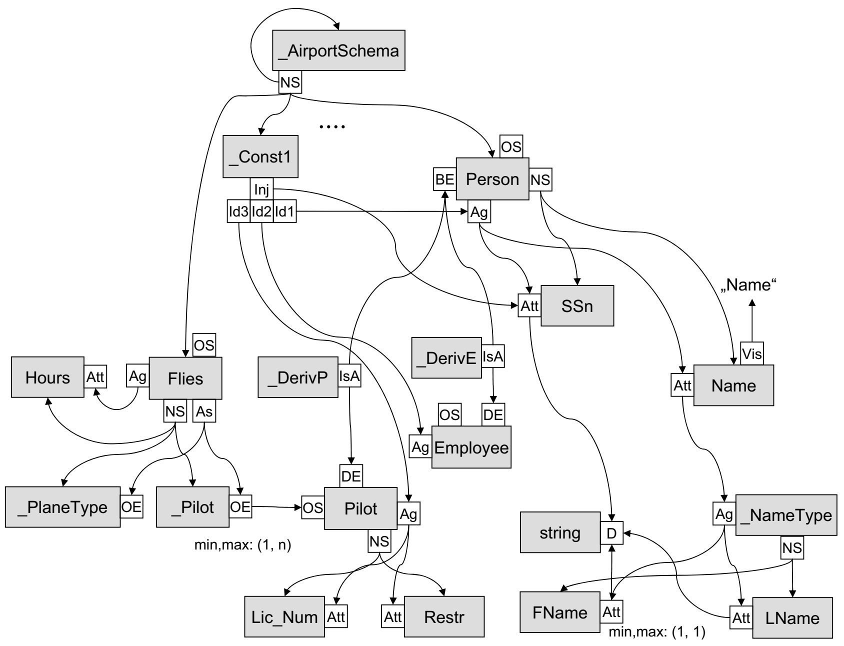

Fig. 4. *GeRoMe* representation of an EER schema

a name. We represent this in the following figures by assigning a simple label to the gray box resembling the element. In case of anonymous elements, which do not play a *Visible* role, we prefix the label with an underscore.

The root model element of the airport schema is a model element representing the schema itself (*_AirportSchema*). It plays a *Namespace* role (NS) referencing all model elements directly contained in this model.

The *Name* attribute is a visible model element and therefore its model element object plays the *Visible* role (Vis). The role defines a name of the element as it could be seen in a graphical EER editor (note that we omitted other occurrences of the *Visible* role class).

Since entity types are composed of attributes, every object representing an entity type plays an *Aggregate* role (Ag). Furthermore, instances of entity types have object identity. Consequently, representations of entity types also play an *ObjectSet* role (OS). The *Aggregate* role is again connected to the descriptions of the entity type's attributes.

The EER model defines a primary key constraint on the *SSn* attribute. Therefore, a model element representing the constraint (*_Const1*) and playing an *Injective* role (Inj) is connected to this attribute. This is a uniqueness constraint which is special in the sense that it has been chosen to be a primary key for the entity type *Person*. This fact is represented by the constraint playing an *Identifier* role (Id1) connected to the identified aggregate. Since *Person*'s subtypes must have the same identifier, the injectiveness constraint plays also *Identifier* roles (Id2, Id3) with respect to these model elements.

In the EER model, it is usually not possible to specify domain constraints, but the addition of default domains does not hurt. Therefore, attributes always have a type in *GeRoMe*. Domains are themselves represented as model elements playing domain roles (D) (e.g. string). It is also possible to derive new types from existing ones as this is also possible in most concrete metamodels.

In addition, note that the composite attribute *Name* has not a domain but another *Aggregate* as type. Unlike the representation of an entity type, *_NameType* is not player of an *ObjectSet* role. Consequently, this element cannot be connected to an *Association-End*, which means that it cannot participate in associations. Furthermore, *_NameType* is not visible as it is an anonymous type. However, the representation is very similar to that of entity types and this eases handling both concepts similarly. For example, in another schema the composite attribute could be modeled by a weak entity type. If these two schemata have to be matched, a generic Match operator would ignore the *ObjectSet* role. The similarity of both elements would nevertheless be recognized as both elements play an *Aggregate* role and have the same attributes.

Furthermore, the figure shows the representation of the *isA* relationship. Since every instance of *Pilot* and *Employee* is also an instance of *Person*, the *Person* model element plays a *BaseElement* role (BE) referenced by two *IsA* roles (IsA). These roles define two children, namely the *DerivedElement* roles (DE) which are played by the respective subtypes *Employee* and *Pilot*. Any attribute attached to the *Aggregate* roles of the subtypes defines an extension to the supertype. The children could also be defined as predicate-defined subtypes by associating to the *DerivedElement* roles a number of *Restriction* roles.

The subtype *Pilot* participates in the relationship type *Flies*. The representation of this relationship contains an *Association* role (As) which is attached to two *Object-AssociationEnd*s (OE) (i.e. a binary relationship). Furthermore, the relationship has an attribute, and consequently, it plays the role of an *Aggregate*. The representations of the two association ends define cardinality constraints and are linked to the *ObjectSet* roles (OS) of their respective participators. They also may play a *Visible* role which assigns a name to the association end.

4.2 Representation of an XML Schema

Figure 6 shows part of an XML Schema for the airport domain whereas figure 5 shows the representation of this example schema in *GeRoMe*. The XML Schema element is a relationship between its enclosing type and the complex type of the nested element. But it is always a 1:n relationship since an XML document is always tree structured. Cross links between elements in different subtrees have to be modeled by references.

But what about root elements in a schema? These elements are related to the schema itself which in our role based model is represented by the *http://../Airport* model element. This is just one example of a concrete model element which is not obviously mapped to a generic metamodel.

An XML document conforming to an XML Schema can have any element as root element which is defined in the schema file as a direct child of the `schema` element. Consequently, any such element is represented in *GeRoMe* as a model element playing an association role with its complex type as one participator and the schema node as

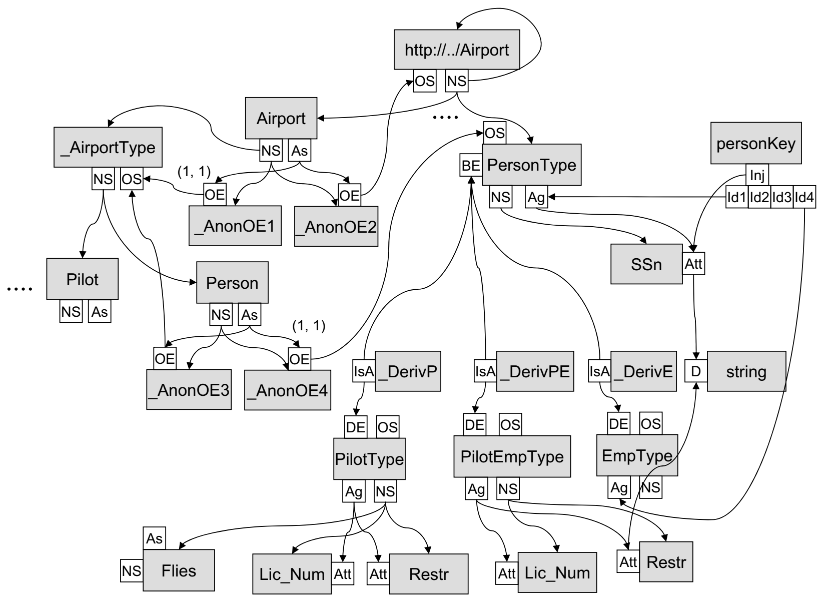

Fig. 5. Representation of a similar XML Schema

the other participator. In the example, *Airport* is the only such element. This element is visible and its name is “Airport”. *AssociationEnd*s of XML elements have no names attached and therefore are anonymous. Complex types may be anonymously nested into an element definition. In the example, this is the case for *_AirportType*. Since definitions of keys have labels in XML Schema, the identifier of *Person* plays a *Visible* role with its label “personKey” assigned to it.

Model elements defined within other model elements such as attributes and XML elements are referenced by the *Namespace* role of the containing element. For example, the element *Flies* is owned by the *Namespace* role of *PilotType*. Another consequence of the structure of semistructured data is that the *AssociationEnd* of the nested type always has cardinality (1, 1), i.e. it has exactly one parent. Finally, the model element *PilotEmpType* has been introduced as it is not possible to represent overlapping types in XML Schema.

4.3 Representation of an OWL DL Ontology

In table 1, we stated that OWL DL object properties are represented by model elements playing *ObjectAssociationEnd* roles and that a pair of these model elements is connected by an *Association*. This is another good example for the problems which occur when integrating heterogenous metamodels to a GMM. The reasons for the sketched representation can be explained with the semantics of the relationship type *WorksOn* in fig. 3. The representation of the corresponding OWL DL elements is shown in figure 7.

Intuitively and correctly, one represents *WorksOn* as a model element playing an *Association* role. *WorksOn* has two *ObjectAssociationEnd*s: one with cardinality (0,n) pointing on *PlaneType* and one with cardinality (1,n) pointing on *Employee*. This is

```
<xsd:schema xmlns="http://../Airport">
  <xsd:element name="Airport">
    <xsd:complexType>..</xsd:complexType>
    <xsd:key name="personKey">
      <xsd:selector xpath="./Person" />
      <xsd:field xpath="@SSn" />
    </xsd:key>
  </xsd:element>
  <xsd:complexType name="PersonType">
    <xsd:attribute name="SSn" type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="PilotType">
    <xsd:complexContent>
      <xsd:extension base="PersonType">
        <xsd:sequence>
          <xsd:element name="Flies">
            <xsd:complexType>...</xsd:complexType>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="Lic_Num" type="xsd:string" />
        <xsd:attribute name="Restr" type="xsd:string" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="PilotEmpType">
    ...
  </xsd:complexType>
</xsd:schema>
```

Fig. 6. An XML Schema for the airport domain

represented analogous to *Flies* in fig. 4. Now what are the problems if you would regard an object property *WorksOn* as corresponding to the given relationship type?

Firstly, an object property always has domain and range. Thus, it has a direction. But the direction of a relationship type is only suggested by its name. On the other hand, an association end has a direction. The role name describes the role which the participator plays in the relationship type with respect to the participator at the opposite end. Furthermore, these role names are often phrasal verbs as are the names of object properties in OWL. Actually, in description logics object properties are often called *roles*. Thus, “WorksOn” should be the role name assigned to the link between the relationship type and the entitiy type *PlaneType*.

Secondly, an object property may have one cardinality restriction, whereas a relationship type has at least two (one for each participating entity). This shows that an object property corresponds to an association end, and that a pair of object properties (one of which is the inverse of the other) is correctly represented as a binary association. Note that OWL DL allows only binary relationships.

In order to allow other constraints, such as *Symmetric*, new roles can be added to *GeRoMe*. Adding a new role to the metamodel will render existing models and operator implementations valid and correct. Thus, it is also easy to extend *GeRoMe* if this is necessary in order to include new modeling constructs.

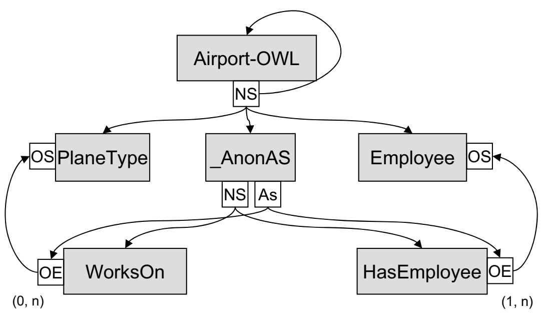

Fig. 7. Representation of OWL object properties

5 Model Management using *GeRoMe* Models

In this section, we show how model management operators can make use of *GeRoMe*. Transformation of models is a typical task for model management applications. Our implementation of generic ModelGen operators is comparable to the approach described in [26] which is imperative as well. Another approach is the rule-based model transformation of [2]. We implemented our Import and Export operators based on equivalence rules between the concrete metamodels and *GeRoMe* (cf. section 6.3). A rule based approach is particularly useful for this task as equivalence rules enable consistent import and export of models since they are applicable in both directions.

We will explain the transformation of the EER model of fig. 4 into a relational schema. Therefore, the original representation has to undergo several transformations in order to become a representation of a relational schema. Fig. 8 shows the final result of the transformation steps which will be discussed in detail in the following.

5.1 Transformation of *GeRoMe* Models

In model management, transformation of models is performed by a ModelGen operator, i.e. the operator generates a model from another existing model. We have implemented the transformation of constructs such as composite attributes or inheritance from an EER schema by several ModelGen_X operators. Each operator transforms the modeling constructs not allowed in the relational model into modeling elements of the relational model. The decomposition of the operators into several "atomic" operators has the advantage that they can be reused in combination with other operators to form new operators. Note that the following operators are not aware about the original representation of the models, i.e. the operators just use the *GeRoMe* representation. Thus, they could also be used to transform a UML model into XML Schema if similar transformation tasks are required (e.g. transformation of associations to references).

It has to be emphasized that mapping of models from one metamodel to another is just one popular example application of model management. The goal of our generic metamodel is *not only* to provide a platform for schema translation but to provide a

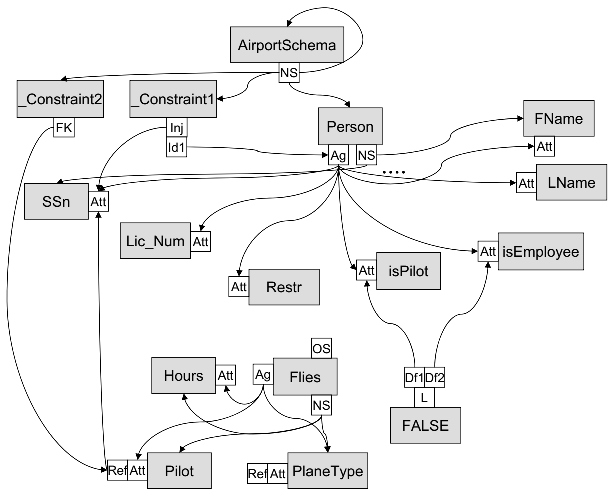

Fig. 8. Representation of the resulting relational schema

generic model representation that serves as a foundation for the polymorphic usage of *any* model management operator. Thereby, other applications of model management, such as schema evolution, are also supported in a generic way.

Transformation of Relationship Types Relationship types are not allowed in the relational metamodel. According to properties such as cardinality constraints, they have to be transformed to relations by executing the operator ModelGen_AssocToRef for each *Association* role. First, it looks for attached *ObjectAssociationEnd* roles, the arity of the association, and cardinality constraints. Depending on these constraints the transformation is either performed automatically or the user is asked for a decision before the operator can proceed. Copies of all attributes in the participators' identifiers are attached to the relationship's *Aggregate* role. An *Aggregate* role has to be created first, if not yet available. Furthermore, these copies play *Reference* roles (Ref) referencing the original attributes, and thereby defining referential constraints. These reference roles constitute a foreign key (FK). After performing all these transformations, the association ends and the relationship's *Association* role are deleted.

The result yet contains *ObjectSet* roles (OS), which are not allowed in a relational model. These roles can now be removed directly, as the associations have been transformed to attribute references. This yields an intermediate result which cannot be interpreted as a valid schema in the EER or relational metamodel, since it now contains constructs disallowed in both metamodels. An Export operator to the Relational or EER metamodel would have to recognize this invalidity and reject to export.

Transformation of IsA Relationships The *isA* relationships also have to be removed depending on their characteristics (partial and overlapping), the attributes of the extensions *Pilot* and *Employee* thereby become attributes of the supertype.

The operator *ModelGen_FlattenIsA* fulfills this task by receiving a *BaseElement* role as input. It first checks for disjointness of connected *isA* relationships and whether they are total or not. Depending on these properties, the user is presented a number of choices on how to flatten the selected *isA* relationships. In the example, the base type *Person* and its subtypes *Pilot* and *Employee* have been selected to be transformed to one single aggregate due to the fact that the *isA* relationship is neither total nor disjoint. The resulting aggregate contains all attributes of the supertype and of the subtypes. Additionally, the boolean attributes *isPilot* and *isEmployee* as well as *Default* roles Df1 and Df2 related to these attributes have been introduced.

Transformation of Composite Attributes The transformation of composite attributes is done by another atomic operator. First, it collects recursively all "atomic" attributes of a nested structure. Then, it adds all these attributes to the original *Aggregate* and removes all the structures describing the composite attribute(s) (including the anonymous type). This operator also needs to consider cardinality constraints on attributes, since set-valued attributes have to be transformed into a separate relation.

In this way, the whole EER schema has been transformed to a corresponding relational schema. Of course, more operators are needed to handle other EER features, such as *Union* derivations of new types.

5.2 Equivalence of Models Represented in Different Metamodels

The preceding sections showed models for the airport domain in different concrete metamodels. It can be seen that, although each of the models is designed for the same domain, their *GeRoMe* representations differ from each other.

Please note that the differences in the representations stem from the constraints and semantics of the concrete metamodels. Nevertheless the representations use the same role classes in all models, while accurately representing the features of the constructs in the concrete modeling languages. For example, the XML Schema *PersonType* plays the same roles as the EER *Person*, since entity types have the same semantics as XML Schema complex types. Furthermore, the relational *Person* does not play the *ObjectSet* and *BaseElement* roles since these are not allowed in the relational model. On the other hand, all these roles play an *Aggregate* role, and therefore they look the same to an operator which is only interested in this role.

In the last section we demonstrated the tranformation of an EER model into a relational model. Because of the aforementioned differences in the semantics of representations in different concrete metamodels a model resulting from such transformations cannot be equivalent to the original model in a formal way. For example, since the relational model does not allow relationship types, these elements have to be transformed to relations with referential constraints. Thus, during the transformation information about the original model is lost because the target metamodel cannot represent these concepts.

Consequently, if you transform the *GeRoMe* representation of an EER model into the *GeRoMe* representation of a relational schema and try to reverse this, the result may be a model which is different from the original schema. For example, it is not possible to identify which model elements stem from entity types or relationship types, respectively.

To summarize, a generic metamodel cannot represent models from different concrete metamodels identically because each concrete metamodel is designed to represent different aspects of real world entities and their relationships. What it can do is to represent models in any metamodel with the same set of modeling elements. This allows to implement model management operators only with respect to these elements of the generic metamodel and to use these operators polymorphically for models from arbitrary metamodels.

6 Architecture and Implementation

Applications dealing with complex models require support for model management in several ways. Therefore, our goal is to provide a library for the management of *GeRoMe* models (including the definition of several operators) that can be reused in various application settings. In the following, we will first present the architecture of our framework. In section 6.3, we will explain how we have realized the import and export of *GeRoMe* models, which is based on a logical formalization presented in section 6.2. More information about the import and export of models and also the transformation of models using a rule based approach in *GeRoMe* can be found in [21].

6.1 Architecture

In order to make the functionalities of *GeRoMe* available to several applications, we developed an API for the manipulation of models represented in *GeRoMe*. Manipulations are performed by a set of model management operators. These can be atomic operators or operators composed of existing operators. Figure 9 presents the structure of our API as well as the general architecture of model management applications based on *GeRoMe*.

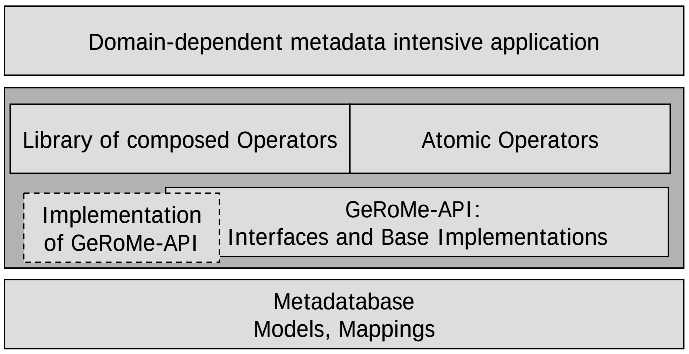
Domain-dependent metadata intensive application
Library of composed Operators
Atomic Operators
Implementation of GeRoMe-API
GeRoMe-API: Interfaces and Base Implementations
Metadatabase
Models, Mappings

Fig. 9. Architecture of a metadata intensive application based on *GeRoMe*

Our implementation of a model management platform is based on a multi-layered architecture. The lowest layer provides facilities to store and retrieve models in the *GeRoMe* representation and is implemented using the deductive metadatabase system ConceptBase [19]. ConceptBase uses Telos as modeling language [25], which allows to represent multiple abstraction levels and to formulate queries, rules and constraints. Objects are represented using a frame-like or graphical notation on the user side, and a logical representation (triples similar to RDF) based on Datalog$^\neg$ internally. The logical capabilities of ConceptBase can be used to analyze the models or to encode the semantics of models in logical rules (e.g. inheritance of attributes)[1]. In addition, *GeRoMe* models can be stored as and constructed from a set of logical facts which is used for the import and export of models. This will be discussed in more detail below. Furthermore, it is possible to store models represented in *GeRoMe* in an XML format to ease the exchange of metadata.

On top of the storage layer, an abstract object model corresponding to the model in fig. 2 has been implemented as a Java library. This is a set of interfaces and base implementations in Java. An implementation of these interfaces can be chosen by instantiating a factory class. Consequently, the object model is independent from the underlying implementation and storage strategy. The relationship between roles and model elements is represented in member variables and methods which allow to add and delete roles from model elements (or other roles). A role has also a link to its player. A model element can be queried for the roles it plays or all roles of a specific class can be retrieved. Some convenience methods in the model element interface allow direct access to the role object (e.g. `getAggregate()`). The *GeRoMe* API is also independent from any original metamodels; the relationship between models represented in *GeRoMe* and models represented in the original metamodels is established by import and export operators as described below in section 6.3.

The next layer is formed by atomic operators based on the *GeRoMe* API. Operators have to be implemented "as atomically as possible" in order to allow maximum reuse. These atomic operators are not aware of the original metamodel, i.e. their implementations use exclusively roles and structures in *GeRoMe*.

By "atomic" we denote that when implementing an operator such as ModelGen_RM, this shall not be implemented as a black box. Instead, the developer should extract certain steps that can be seen as meaningful units of manipulation and can be reused in other tasks. Such an atomic operator should not implicitly encode any knowledge about the native metamodel it operates on.

Atomic operator implementations can be combined to form new, more complex, composite operators. In doing so, the reuse of operators is increased in two ways. On the one hand, operators have to be implemented only once for the generic metamodel and can be used for different concrete metamodels.

For example an operator for the transformation of association ends to referential constraints can be reused by a composite operator *ModelGen_RM* which computes a relational model from an EER model and by another composite operator *ModelGen_XS* for computing an XML Schema from an OWL ontology.

On the other hand, operators such as Match or Merge can be reused to compose new operators for solving different metadata related tasks. Thus, a metadata intensive application uses atomic and composite operators to implement its model manipulation functionality.

[1] We are aware of the fact that Datalog cannot be used to support full reasoning on all modeling languages, especially not OWL DL. However, this is not the goal of *GeRoMe*, it is used to represent the explicit knowledge about models which will be used in model management operators. Full reasoning about models has still to be done by special purpose reasoners for the specific metamodels.

6.2 Logical Formalization

A logical formalization of *GeRoMe* enables the specification of several model management tasks in a declarative way. As we will describe in the next section, import and export of models can be easily defined by rules using such a formalization. Furthermore, this logical representation enables also more sophisticated reasoning mechanisms on models, for example, to check the consistency of models or the correctness of transformations.

Formally, *GeRoMe* is defined by a set of role types $\mathcal{R} = \{r_1, \ldots, r_n\}$ and a set of attribute types $\mathcal{A} = \{a_1, \ldots, a_m\}$ which can be applied to role types. In addition, $\mathcal{V}$ denotes a set of atomic values which may be used as attribute values. A model M represented in *GeRoMe* is defined by a tuple $M = \langle E, R, type, plays, attr \rangle$, where

- $E = \{e_1, \ldots, e_k\}$ is a set of model elements,
- $R = \{o_1, \ldots, o_p\}$ is a set of role objects,
- $type : R \rightarrow \mathcal{R}$ is a total function that assigns exactly one role class to each role object,
- $plays \subseteq (E \cup R) \times R$ represents the aforementioned relation between model elements (or role objects) and role objects,
- $attr \subseteq (R \times \mathcal{A}) \times (R \cup \mathcal{V})$ represents the attribute values of a role object (i.e. attributes may also refer to other role objects).

To make the representation more human-readable, we have used a simplified notation in the formulation of the import/export rules below. The fact that an object e is a model element ($e \in E$) is represented by the statement `modelElement(e)`. 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, `objectAssociationEnd(e)` states that the model element `e` plays the *ObjectAssociationEnd* 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, `min(objectAssocationEnd(e),1)` specifies that the *min*-attribute of the role object defined above is 1.

We have also defined a formal semantics for *GeRoMe* to characterize data instances (see appendix A.1), which is in line with the logical formalization given above. Data instances are also used at the model level (see role *Instance*), e.g. as default values or boundaries of a type defined by an interval. The main goal of the formal semantics is however the formal definition of mappings between models, which should finally be used to translate data instances from one model to another model. As this is out of the scope of this paper, we do not elaborate on the formal semantics here.

Using the logical representation for *GeRoMe* models and a similar representation for models in specific metamodels, we can use a rule-based approach for the import and export of models as we will present in the next section. Moreover, this representation of a model is a fine grained representation, because each feature (or property) of a model element is represented by a separate fact. This is especially useful for the Diff operator in which we need to identify the differences of model elements.

6.3 Import and Export Operators

We do not continuously synchronize a *GeRoMe* model with underlying native metadata. Instead, we import the native metadata into *GeRoMe* and after manipulating this model we export it into some native format. A process that has also been used in [3]. In general, it is not even possible to enforce consistency of the native schema with the *GeRoMe* model since manipulations may yield an intermediate result that is not valid in neither the source nor the target modeling language. Consider the example of section 5. An EER model in which some of the relationship types have been transformed into foreign keys but others have not, is neither a valid EER model, as it contains references, nor a valid relational schema as it still contains relationship types. Consequently, the *GeRoMe* model cannot be synchronized with a native schema. Instead, only the input and output must be representations of valid native models.

Import and export operators to the native format of the various modeling languages are currently being implemented. The operators use the logical representation presented before and a rule-based approach: the relationship between a concrete metamodel and *GeRoMe* is represented by a set of equivalence rules. The left hand side of a rule refers to elements of the concrete metamodel, the right hand side refers to *GeRoMe* elements.

Using a rule-based approach for specifying the import/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. Furthermore, 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 import/export operators significantly.

Formally, a *GeRoMe* model is represented by a set of ground facts KB_{GeRoMe} which uses only vocabulary (functions, predicates, ..) from the logical *GeRoMe* representation (e.g. $modelElement(Person)$, $attribute(Person, Name), \ldots$). The model itself corresponds to the one and only logical model M of KB_{GeRoMe}. This interpretation is trivial but it has to be noted that there must be only one logical model, otherwise KB_{GeRoMe} is ambigious. This is one requirement that has to be considered when implementing the rules for import and export.

Now, a model in a concrete metamodel (say EER) is also represented by a set of ground facts KB_{EER} about model elements which uses only vocabulary from the concrete metamodel ($UMLAssociationEnd(as, ae, rn, min, max)$, $RMTable(x)$, $EEREntityType(\ldots), \ldots$). The import amounts to applying a set of rules to the facts KB_{EER} (say $S_{EER \leftrightarrow GeRoMe}$). The left hand side of the implication contains only vocabulary from the concrete metamodel, the right hand side contains only vocabulary from *GeRoMe*. The result is a set of ground facts KB_{GeRoMe} (instantiations of the right hand sides). The export of a model is performed the other way around; to have consistent import and export operators, the rules are expressed as equivalence rules which can be interpreted from left to right or vice versa.

Fig. 10 gives an example for such rules. The rules are expressed in standard Prolog syntax, i.e. labels starting with an upper-case letter denote variables. They are evaluated using a meta-program implemented in Prolog, which is able to handle rules with mul-

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

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

Fig. 10. Example rules for the Import/Export of SQL models

tiple predicates on both sides of the equivalence. 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 domain 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. The second rule represents the special case in which NULL-values are allowed for the column, which is represented in *GeRoMe* by a minimum cardinality of 0.

In the example of Fig. 10 we have used terms as arguments of some predicates (e.g. `namespace(TableID)`). As described above, these terms represent the role objects. With a pure logical view, one could also interpret these terms just syntactically as *Skolem functions*, which have been introduced on the right hand side to replace existentially quantified variables, i.e. variables that would appear only on one side of the rule. As the goal is to construct objects using the *GeRoMe*-API, these functions must return meaningful objects. Therefore, while creating the *GeRoMe* objects from a set of facts, these functions will return the corresponding role objects of the given model elements, e.g. `namespace(TableID)` returns the *Namespace* role of the model element `TableID`. By doing so, we make sure that the same objects are used, even if they are referenced in different rules; for example, the attribute role of `ID` is referenced in both rules of fig. 10. Note that in some cases, objects can play multiple roles of the same type (e.g. attributes may play several reference roles); in this case, these functions have more than one argument (i.e. all objects that are necessary to identify the role).

Fig. 11 presents an example of import/export rules for complex types of XML schemas. The first part of the right hand side of the rule is similar to the example before; it defines a model element which is contained in a namespace and which plays a visible role. In addition, the model element plays also the *ObjectSet* role, as complex types participate in associations. The second rule adds an aggregate rule to the model element

```
xs_namespace(NamespaceID),
xs_complextype(ID),
xs_complextype_ns(ID, NamespaceID),
xs_complextype_name(ID, Name) <=>
   modelElement(ID),
   owned(namespace(NamespaceID),ID),
   visible(ID),
   name(visible(ID),Name),
   objectSet(ID).

xs_attribute(ID),
xs_attribute_of(ID, ComplexTypeID) <=>
   aggregate(ComplexTypeId),
   modelElement(ID),
   attribute(ID),
   property(aggregate(ComplexTypeID),attribute(ID)),
   ...
```

Fig. 11. Example rules for the Import/Export of XML Schemas

of this complex type, if the complex type contains also attributes. The rule creates also a model element (`ID`) for the attribute and links the attribute role of this object to the aggregate role of the complex type. We omitted further statements for the definition of namespaces, etc.

Note that the rules can be used in both ways. Thus, it is also possible to export *GeRoMe* models using these rules. Depending on the desired target metamodel, the corresponding rule set will be activated and evaluated based on a set of facts representing the *GeRoMe* model. Evaluating the rules is only one step of the export operator: before a model can be exported to a concrete metamodel, the export operator has to check whether all roles used can be represented in the target metamodel. If not, the problematic roles have to be transformed into different elements as described, for example, in section 5.

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. Since the correspondences are not hidden in imperative code, but are given as a set of equivalence rules, the developer can concentrate on the logical correspondences and does not have to deal with implementation details. Besides, only two classes have to be implemented that produce facts about a concrete model from an API (e.g., the Jena OWL API, see fig. 12) or read facts and produce the model with calls to the API, respectively. These two classes merely produce (or read) a different syntactic representation of the native model and do not perform any sophisticated processing of schemas. Creating and processing of facts about the *GeRoMe* representation is completely done with reflection. During the export, facts about a *GeRoMe* model are created according to Java annotations in the API. During the import unary facts cause model elements and role objects to be created, binary facts establish relationships between objects and ternary facts do the same for indexed relationships (see fig. 13). This significantly reduces the programming effort for supporting a new metamodel. 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

```
public String transformClass(OntClass cls) {
  //...
  List<OntClass> lClasses=cls.listSuperClasses(true).toList();

  for(OntClass superClass : lClasses) {
    sResID = transformClass(superClass);
    Term t=Prolog.term("owl_subclass",plID,Prolog.id(sResID));
    mlFacts.add(t);
  }
}
```

Fig. 12. Example code fragement for importing OWL classes

```
// term is a Java object representing a Prolog term
if(term.arity() == 1) {
  if(!invokeBuilder(term)) {
    throw new ModelManException("No such method");
  }
}
else if(term.arity() == 2 || term.arity() == 3) {
  if(!invokeMethod(term)) {
    throw new ModelManException("No such property");
  }
}
```

Fig. 13. Creating *GeRoMe* objects using reflection

Prolog rules. The relationships between the modeling constructs could be expressed in less than 20 equivalence rules.

6.4 Equivalence of Imported and Reexported Models

The transformations, performed by ModelGen operators such as the ones presented in section 5, in general serve the purpose of removing constructs disallowed in the target metamodel. Therefore, the transformation cannot be reversed automatically as it removed information from the original model which can only be regained by asking the user. At best, suggestions can be made based on heuristics.

However, the import and subsequent export for a generic metamodel should not lose information. It must be emphasized that an import to and an export from *GeRoMe* may result in a model syntactically 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 must 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.

On the other hand, semantic equivalence of imported and subsequently reexported models means that the same set of instances (individuals, tuples, XML fragments, ..) satisfies both, the original model and the imported and reexported model. The mapping rules for metamodels described above should be formulated in a way which ensures that this property holds.

Equivalence between models can be defined by means of information capacity [16,24] This definition must be adopted in our case to metamodels. Let f be a mapping between the native metamodel M and *GeRoMe* defined by a set of mapping rules R. A subsequent import and export can only yield the original model if f is invertible, so f^{-1} and f can be composed. Therefore, it must be a total and injective function from the set of valid models in M to the set of valid $GeRoMe$ models. Then f is an *information capacity preserving mapping* [16,24] between the sets of models, and *GeRoMe* dominates M via f and, naturally, the composition of f and f^{-1} is the identity function (an *equivalence preserving mapping*) on the set of models in metamodel M. Consequently, the above notion of semantic equivalence would be satisfied.

Thus, the question of whether a model in a native metamodel can be losslessly imported and reexported can only be answered with respect to the allowed modeling constructs and the mapping rules R for the respective metamodel. These mapping rules must translate every native modeling construct uniquely into a corresponding generic modeling construct (or combination thereof) and translate the same generic construct into the same native modeling construct. Indeed, there are some constructs, that still cannot be represented in *GeRoMe*. For instance, as we concentrated on data models, we cannot model methods in *GeRoMe*. But *GeRoMe* is designed to be extendable; if it is not possible to represent a modeling construct in the correct way in *GeRoMe*, new roles can be added to do so. We have made this experience while implementing the mapping rules for XML Schema; it contains several modeling features which are not available in other modeling languages. For instance, the *LiteralAssociationEnd* role has been introduced to model XML elements with simple type. These could as well be modeled as attributes, but then it would not be possible to tell whether an attribute should be exported to an XML Schema attribute or an element.

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. 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 [5]. 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.

7 Conclusion

Generic model management requires a generic metamodel to represent models defined in different modeling languages (or metamodels). The definition of a generic metamodel is not straightforward and requires the careful analysis of existing metamodels. In this paper, we have presented the generic role based metamodel *GeRoMe*, which is based on our analysis and comparison of five popular metamodels (Relational, EER, UML, OWL, and XML Schema).

We recognized that the intuitive approach of identifying generic metaclasses and one-to-one correspondences between these metaclasses and the elements of concrete metamodels is not appropriate for generic metamodeling. Although classes of model elements in known metamodels are often similar, they also inhibit significant differences which have to be taken into account. We have shown that role based metamodeling can be utilized to capture both, similarities and differences, in an accurate way while avoiding sparsely populated intersection classes. In addition, the role based approach enables easy extensibility and flexibility as new modeling features can be added easily. Implementations of operators access all roles they need for their functionality but remain agnostic about any other roles. This reduces the complexity of models from an operator's point of view significantly. Furthermore, the detailed representation of *GeRoMe* models is used only by a model management application, users will still use their favorite modeling language.

Whereas role based modeling has yet only been applied to the model level, we have shown that a generic metamodel can benefit from roles. In particular, *GeRoMe* enables *generic* model management. As far as we know, the role based approach to the problem of generic metadata modeling is new. It has been validated by representing several models from different metamodels in *GeRoMe*.

We implemented a framework for the management of models including an object model for *GeRoMe* models that allows operators to manipulate, store, and retrieve models. Atomic model management operators are implemented based on our generic metamodel and can be combined to composite operators. In particular, the usage of a generic metamodel allows to apply operator implementations polymorphically to models represented in various modeling languages which increases the reusability of operators. As a first evaluation of *GeRoMe*, we have implemented some ModelGen operators.

We have developed a *rule-based* approach for import and export operators which is based on a logical formalization of *GeRoMe* models. These operators will also be used to verify that the model elements of different metamodels can be represented accurately and completely in *GeRoMe*.

Future work will concentrate on the development of further model management operators. We have started working on the implementation of a Match operator for *GeRoMe* models and are investigating how the generic representation can be exploited by this operator. We have defined a formal semantics for *GeRoMe* which was necessary to describe the structure of instances of *GeRoMe* models. The semantics is also used for a formal definition of mappings between *GeRoMe* models which we are currently designing. The mapping representation will be used by model management operators such as Merge and Compose.

While it might be necessary to integrate new modeling features of other languages, or features which we did not take into account so far, we are confident that our work is a basis for a generic solution for model management.

Acknowledgements: This work is supported in part by the EU-IST project SEWASIE (`http://www.sewasie.org`) and the EU Network of Excellence ProLearn (`http://www.prolearn-project.org`).

References

1. P. Atzeni, P. Cappellari, P. A. Bernstein. A Multilevel Dictionary for Model Management. L. M. L. Delcambre, C. Kop, H. C. Mayr, J. Mylopoulos, O. Pastor (eds.), *Proc. 24th Inter-*

national Conference on Conceptual Modeling (ER), *Lecture Notes in Computer Science*, vol. 3716, pp. 160–175. Springer, Klagenfurt, Austria, 2005.

2. P. Atzeni, P. Cappellari, P. A. Bernstein. Model-Independent Schema and Data Translation. Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, C. Böhm (eds.), *Proc. 10th International Conference on Extending Database Technology (EDBT)*, *Lecture Notes in Computer Science*, vol. 3896, pp. 368–385. Springer, Munich, Germany, 2006.
3. P. Atzeni, R. Torlone. Management of Multiple Models in an Extensible Database Design Tool. P. M. G. Apers, M. Bouzeghoub, G. Gardarin (eds.), *Proc. 5th International Conference on Extending Database Technology (EDBT)*, *Lecture Notes in Computer Science*, vol. 1057, pp. 79–95. Springer, Avignon, France, 1996.
4. C. W. Bachman, M. Daya. The Role Concept in Data Models. *Proceedings of the Third International Conference on Very Large Data Bases (VLDB)*, pp. 464–476. IEEE-CS and ACM, Tokyo, Japan, 1977.
5. D. Barbosa, J. Freire, A. O. Mendelzon. Information Preservation in XML-to-Relational Mappings. Z. Bellahsene, T. Milo, M. Rys, D. Suciu, R. Unland (eds.), *Proc. of the Second International XML Database Symposium, XSym 2004, Toronto, Canada*, *Lecture Notes in Computer Science*, vol. 3186, pp. 66–81. Springer, August 2004.
6. M. Baumeister, M. Jarke. Compaction of Large Class Hierarchies in Databases for Chemical Engineering. *8. GI-Fachtagung fr Datenbanksysteme in Bro, Technik und Wissenschaft (BTW)*, pp. 343–361. Springer, Freiburg, 1999.
7. P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. *Proc. First Biennial Conference on Innovative Data Systems Research (CIDR2003)*. Asilomar, CA, 2003.
8. P. A. Bernstein, A. Y. Halevy, R. Pottinger. A Vision for Management of Complex Models. *SIGMOD Record*, **29**(4):55–63, 2000.
9. P. A. Bernstein, S. Melnik, P. Mork. Interactive Schema Translation with Instance-Level Mappings. K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, B. C. Ooi (eds.), *Proc. 31st International Conference on Very Large Data Bases (VLDB)*, pp. 1283–1286. ACM Press, Trondheim, Norway, 2005.
10. P. A. Bernstein, S. Melnik, M. Petropoulos, C. Quix. Industrial-Strength Schema Matching. *SIGMOD Record*, **33**(4):38–43, 2004.
11. E. Bertino, G. Guerrini. Objects with Multiple Most Specific Classes. *Proc. European Conference on Object-Oriented Programming (ECOOP)*, *Lecture Notes in Computer Science (LNCS)*, vol. 952, pp. 102–126. Springer, Aarhus, Denmark, 1995.
12. R. A. Elmasri, S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, Reading, Mass., 3rd edn., 1999.
13. R. Fagin, P. G. Kolaitis, L. Popa, W. C. Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Transactions on Database Systems*, **30**(4):994–1055, 2005.
14. S. Hanenberg, R. Unland. Roles and Aspects: Similarities, Differences, and Synergetic Potential. *Proc. 8th International Conference on Object-Oriented Information Systems*, *Lecture Notes in Computer Science (LNCS)*, vol. 2425, pp. 507 – 520. Springer, Montpellier, France, 2002.
15. M. A. Hernández, R. J. Miller, L. M. Haas. Clio: A Semi-Automatic Tool For Schema Mapping. *Proc. ACM SIGMOD Intl. Conference on the Management of Data*, p. 607. ACM Press, Santa Barbara, CA, 2001.
16. R. Hull. Relative Information Capacity of Simple Relational Database Schemata. *SIAM Journal of Computing*, **15**(3):856–886, August 1986.
17. R. Hull, R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, **19**(3):201–260, 1987.
18. ISO/IEC. Information technology – Information Resource Dictionary System (IRDS) Framework. *International Standard ISO/IEC 10027:1990*, DIN Deutsches Institut für Normung, e.V., 1990.
19. M. A. Jeusfeld, M. Jarke, H. W. Nissen, M. Staudt. ConceptBase – Managing Conceptual Models about Information Systems. P. Bernus, K. Mertins, G. Schmidt (eds.), *Handbook on Architectures of Information Systems*, pp. 265–285. Springer-Verlag, 1998.

20. M. A. Jeusfeld, U. A. Johnen. An Executable Meta Model for Re-Engineering of Database Schemas. *Proc. 13th Intl. Conference on the Entity-Relationship Approach (ER94)*, *Lecture Notes in Computer Science (LNCS)*, vol. 881, pp. 533–547. Springer-Verlag, Manchester, U.K., 1994.
21. D. Kensche, C. Quix. Transformation of Models in(to) a Generic Metamodel. submitted for publication.
22. M. Lenzerini. Data Integration: A Theoretical Perspective. L. Popa (ed.), *Proceedings of the Twenty-first ACM Symposium on Principles of Database Systems (PODS)*, pp. 233–246. ACM Press, Madison, Wisconsin, 2002.
23. S. Melnik, E. Rahm, P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pp. 193–204. ACM, San Diego, CA, 2003.
24. R. J. Miller, Y. E. Ioannidis, R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. R. Agrawal, S. Baker, D. A. Bell (eds.), *Proc. 19th International Conference on Very Large Data Bases (VLDB)*, pp. 120–133. Morgan Kaufmann, Dublin, Ireland, 1993.
25. J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis. Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, **8**(4):325–362, 1990.
26. P. Papotti, R. Torlone. Heterogeneous Data Translation through XML Conversion. *Journal of Web Engineering*, **4**(3):189–204, 2005.
27. R. Pottinger, P. A. Bernstein. Merging Models Based on Given Correspondences. J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, A. Heuer (eds.), *Proc. 29th International Conference on Very Large Data Bases (VLDB)*, pp. 826–873. Morgan Kaufmann, Berlin, Germany, 2003.
28. E. Rahm, P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, **10**(4):334–350, 2001.
29. J. Richardson, P. Schwarz. Aspects: extending objects to support multiple, independent roles. *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pp. 298–307. Denver, CO, 1991.
30. R. K. Wong, H. L. Chau, F. H. Lochovsky. A Data Model and Semantics of Objects with Dynamic Roles. *Proc. 13th Intl. Conference on Data Engineering (ICDE)*, pp. 402–411. IEEE Computer Society, Birmingham, UK, 1997.

A Appendix

A.1 Formal Semantics of *GeRoMe*

The formalization of *GeRoMe* in section 6.2 described how *GeRoMe* models can be represented as a set of logical facts, and how this representation can be used to implement the import and export operators.

To describe the semantics of a *GeRoMe* model, we have to characterize what are the valid instances of a *GeRoMe* model. For example, we have to define for a model element $Person$ how intances of this model element may look like and which are valid relationships between persons and other instances. In the following, we will first define the formal semantics of *GeRoMe*, and a simplified notation for instances of *GeRoMe* models that is used in data mappings.

Semantics

Definition 1 (Atoms and Object Identifiers).

– *A denotes a set of atoms, which are literal values of simple datatypes, e.g. "HLX", "Boeing-747", "John", "Smith", "5.2".*

– *O denotes a set of object identifiers, which are used to distinct two instances with the same component values from each other if they are instances of an $ObjectSet$.*

Definition 2 (*GeRoMe* Interpretation). *An interpretation $\mathfrak{I}$ in* GeRoMe *is a tuple $\mathfrak{I} =< \mathcal{I}, \mathcal{O}, \mathcal{P}, \mathcal{V}, A, O, \epsilon >$ where O is a set of object identifiers and A is a set of atoms (literals, ..) as defined before. $\epsilon \notin O \cup A$ is the null value, which can be used for null data values (for attributes), null participators (for association ends), or null object identifiers (for elements* without *object identity).*

$\mathcal{I}$ is the interpretation mapping which maps a model element (specifying a data set, not a constraint) to a set of instances.
$\mathcal{O}$ maps a model element to the set of its object identifiers.
$\mathcal{P}$ maps a model element to a set of Association *instantiations.*
$\mathcal{V}$ maps a model element to a set of data values.
The interpretation mapping is defined as follows:

– *If m is a* Domain *then $\mathcal{I}[m] \subseteq A$ is the set of atoms in m.*
– *If m is an* ObjectSet*, an* Association*, an* Aggregate*, or any combination of these then*
  $\mathcal{I}[m] \subseteq [\mathcal{O}[m] \times \mathcal{V}[m] \times \mathcal{P}[m]]$ where
  - *If m is an* ObjectSet *then $\mathcal{O}[m] \subseteq O$ is the set of object identifiers of instances of m, otherwise $\mathcal{O}[m] = \{\epsilon\}$.*
  - *If m is an* Association *with* AssociationEnd*s AE_i $i = 1, \ldots, n$ then*
    $\mathcal{P}[m] \subseteq [(\mathcal{O}[AE_1.participator] \cup \epsilon) \times \ldots \times (\mathcal{O}[AE_n.participator] \cup \epsilon)]$, otherwise $\mathcal{P}[m] = \{\epsilon\}$. The participator *of an* AssociationEnd *is always an* ObjectSet*. Consequently, this defines a set of tuples of object identifiers. If an association end may be null, ϵ may be the value of this participator. If an association end may participate more than once, each participation is instantiated by another tuple, that is, another element of $\mathcal{I}[m]$. Thus, multiple participations of one participator are multiple instances of the association.*
  - *If m is an* Aggregate *with* Attribute*s A_i $i = 1, \ldots, n$ then*
    $\mathcal{V}[m] \subseteq [\mathbb{P}(\mathcal{I}[A_1.type]) \times \ldots \times \mathbb{P}(\mathcal{I}[A_n.type])]$,
    otherwise $\mathcal{V}[m] = \{\epsilon\}$. Infinite, recursive structures are not allowed, i.e. an element x of $\mathcal{V}[m]$ must not contain any element in which x occurs. The cardinality of a component of $\mathcal{V}[m]$ must be within the (min,max) constraints of the attribute.

Examples Figure 14 gives an example schema containing various combinations of the *ObjectSet*, *Asssociation* and *Aggregate* roles. A possible interpretation are the following: $O = \{1, 2, 3\}$,
$A = \{"InsuranceCorp.", "0815", "John", "Smith", 2500.00\}$,
$\mathcal{O}[Company] = \{1\}$,
$\mathcal{O}[Person] = \{2\}$,
$\mathcal{O}[Employment] = \{3\}$,
$\mathcal{O}[\_theAnonType] = \{\epsilon\}$,
$c \in \mathcal{I}[Company]$ with $c =< 1, \epsilon, < "InsuranceCorp." >>$,
$p \in \mathcal{I}[Person]$ with $p =< 2, \epsilon, < "0815", < \epsilon, \epsilon, < "John", "Smith >>>$,
$n \in \mathcal{I}[\_theAnonType]$ with $c =< \epsilon, \epsilon, < "John", "Smith" >>$,
$e \in \mathcal{I}[Employment]$ with $c =< 3, < 1, 2 >, < 2500.00 >>$,

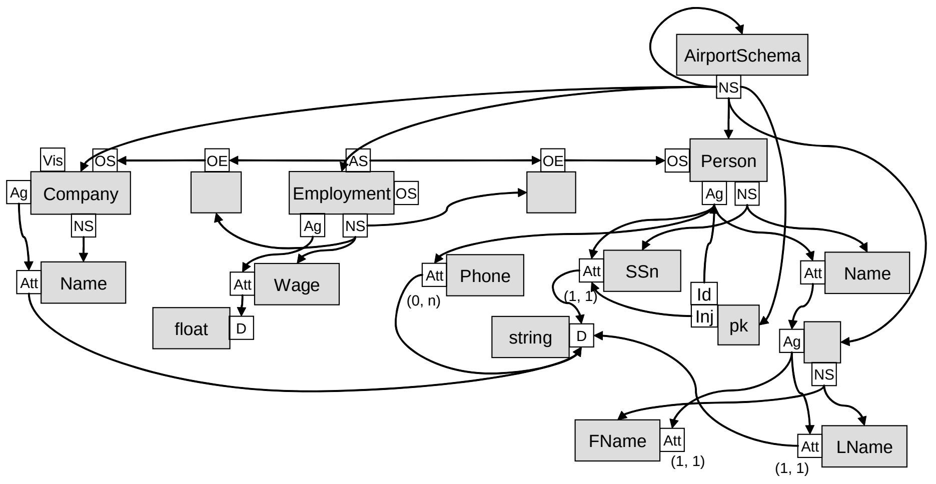

Fig. 14. An example schema about persons and companies

Simplified Notation For the formulation of constraints, queries, and mappings, we choose a simpler representation that uses flat logical predicates instead of the complex terms used before.

Definition 3 (Notation of instances as logical facts). *The interpretation $\mathfrak{I}$ of a model M is represented by a set of facts $\mathcal{D}_M$ as described below.*

- *The interpretation of a model element $\mathcal{I}[m]$ is represented by a set of abstract identifiers $\{id_1, \ldots, id_n\}$. The set of all abstract identifiers is denoted by $\mathcal{T}$. $inst(id_i, m) \in \mathcal{D}_M$ means that the object represented by id_i is an instance of m.*
- *$\forall$ model elements m playing a Domain role and $\forall v \in \mathcal{I}[m] : value(id_i, v) \in \mathcal{D}_M$ and $inst(id_i, m) \in \mathcal{D}_M$.*
- *$\forall$ model elements m playing an ObjectSet role and $\forall o \in \mathcal{O}[m]$: $oid(id_i, o) \in \mathcal{D}_M$ and $inst(id_i, m) \in \mathcal{D}_M$. Each id_i has at most one object identifier o, and each object identifier o is related to exactly one id_i. There is no $oid(id_i, o) \in \mathcal{D}_M$ with $o = \epsilon$.*
- *$\forall$ model elements m playing an Aggregate role and having the attribute a (model element), and this instance has the value $v \in \mathcal{T}$ for that attribute: $attr(id_i, a, v) \in \mathcal{D}_M$ and $inst(id_i, m) \in \mathcal{D}_M$.*
- *$\forall$ model elements m playing an Association role in which the object with identifier o participates for the association end ae: $part(id_i, ae, o) \in \mathcal{D}_M$ and $inst(id_i, m) \in \mathcal{D}_M$.*
- *There are no other elements in $\mathcal{D}_M$.*

Please note that the existence of a predicate like $attr(id, a, v)$ or $part(id, ae, o)$ in $\mathcal{D}_M$ requires the existence of other predicates in $\mathcal{D}_M$ to assure a consistent model (e.g., an attribute value has to be an instance of the type of that attribute).

The "artificial" identifiers id_i are introduced here to reify the complex tuples of an interpretation $\mathfrak{I}$ in order to have flat tuples.

Example The example given above is represented by the following set of facts ($\mathcal{T} = \{\#1, \#2, \#3, \ldots\}$):

$$\begin{array}{ll}
oid(\#1, 1) & oid(\#2, 2) \\
oid(\#3, 3) & \\
attr(\#1, Name, \#5) & attr(\#2, SSN, \#6) \\
value(\#5, "InsuranceCorp") & value(\#6, "0815") \\
attr(\#2, Name, \#4) & attr(\#4, FName, \#7) \\
 & value(\#7, "John") \\
attr(\#4, LName, \#8) & attr(\#3, Wage, \#9) \\
value(\#8, "Smith") & value(\#9, "2500") \\
part(\#3, EmployedBy, \#1) & part(\#3, Employs, \#2) \\
inst(\#1, Company) & inst(\#2, Employee) \\
inst(\#3, Employment) & inst(\#4, \_theAnonType) \\
inst(\ldots) \text{ for values} &
\end{array}$$

A.2 Queries and Mappings in *GeRoMe*

Using the formal semantics of *GeRoMe* models, it is straightforward to represent formulas over *GeRoMe* models that can be used as queries or mappings.

A query in *GeRoMe* is a conjunctive query using the predicates defined above in definition 3. A mapping is basically a relationship of queries over two different models. As it has been proven in [13], mappings expressed second-order tuple generating dependencies (SO tgds) are closed under composition, but first-order tgds are not. Therefore, we use SO tgds to express mappings between models.

Definition 4 (*GeRoMe* model mapping). *A* GeRoMe *model mapping (or, in short, mapping) is a triple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, where $\mathbf{S}$ and $\mathbf{T}$ are the source model and the target model respectively, and Σ is a finite set of formulas of the form*

$$\exists \mathbf{f}((\forall \mathbf{x_1}(\varphi_1 \rightarrow \psi_1)) \wedge \ldots \wedge (\forall \mathbf{x_n}(\varphi_n \rightarrow \psi_n)))$$

where each member of $\mathbf{f}$ is a function symbol, and where each φ_i is a conjunction of atomic formulas and/or equalities over $\mathbf{S}$ and ψ_i is a conjunction of atomic formulas over $\mathbf{T}$ as defined in definition 3. Furthermore, the variables of $\mathbf{x_i}$ appear in at least one atomic formula of φ_i.

The predicates from definition 3 can also be used in first-order logic formulas to express constraints on models. For example, the following formula states that employees working at "Insurance Corp." earn more than 2000 EUR.

$$\begin{array}{l}
\forall x, y, z, v, n \quad inst(x, Employment) \wedge attr(x, Wage, y) \wedge value(y, v) \wedge \\
part(x, EmployedBy, z) \wedge attr(z, Name, n) \wedge n = "InsuranceCorp." \\
\Rightarrow v > 2000
\end{array}$$

As it would be very inefficient to transform data into the *GeRoMe* representation, it is not intended that these queries, mappings, and constraints are actually evaluated on the *GeRoMe* models. Instead, these expressions will be translated into the native query format of the original metamodel (e.g. SQL for a relational database schema) and executed by the specific query evaluation engines.