Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages*

Gerti Kappel[1], Elisabeth Kapsammer[2], Horst Kargl[1], Gerhard Kramler[1], Thomas Reiter[2], Werner Retschitzegger[2], Wieland Schwinger[3], and Manuel Wimmer[1]

[1] Business Informatics Group, Vienna University of Technology
{gerti, kargl, kramler, wimmer}@big.tuwien.ac.at
[2] Information Systems Group, Johannes Kepler University Linz
{ek, tr, wr}@ifs.uni-linz.ac.at
[3] Dept. of Telecooperation, Johannes Kepler University Linz
wieland.schwinger@jku.at

Abstract. The use of different modeling languages in software development makes their integration a must. Most existing integration approaches are metamodel-based with these metamodels representing both an abstract syntax of the corresponding modeling language and also a data structure for storing models. This implementation specific focus, however, does not make explicit certain language concepts, which can complicate integration tasks. Hence, we propose a process which semi-automatically lifts metamodels into ontologies by making implicit concepts in the metamodel explicit in the ontology. Thus, a shift of focus from the implementation of a certain modeling language towards the explicit reification of the concepts covered by this language is made. This allows matching on a solely conceptual level, which helps to achieve better results in terms of mappings that can in turn be a basis for deriving implementation specific transformation code.

1 Introduction

The shift from code-centric to model-centric software development places models as first-class entities in model-driven development processes. A rich variety of modeling languages and tools are available supporting development tasks in certain domains. Consequently, the exchange of models among different modeling tools and thus the integration of the respective modeling languages becomes an important prerequisite for effective software development processes. Due to a lack of interoperability, however, it is often difficult to use tools in combination, thus the potential of model-driven software development cannot be fully exploited.

In collaboration with the Austrian Ministry of Defense and based on experiences gained in various integration scenarios, e.g., [17], [27] we are currently realizing a system called *ModelCVS* which aims at enabling tool integration through transparent transformation of models between metamodels representing different tools' modeling

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806.

O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 528 – 542, 2006.

languages. However, metamodels typically serve as an abstract syntax of a modeling language and often also as an object-oriented data structure in which models are stored. A direct integration of different modeling languages by their metamodels is not a trivial task, and often leads to handcrafted solutions created in an error-prone process usually inducing high maintenance overheads. The integration can be made easier, when concentrating on the concepts described by a language, only, without needing to worry how the language implements these concepts. Geared towards capturing knowledge in a certain domain, *ontologies* can help to explicitly represent the concepts of a language, and thus concentrate the integration task on a solely conceptual level. Furthermore, ontologies enable tasks like logical reasoning and instance classification that can yield additional benefits for semantic integration.

In accordance with the general understanding of the term, we refer to the process of preparing a modeling language for such integration on a conceptual level as *lifting*, which allows to transform a metamodel (abstract syntax) into an ontology representing the concepts covered by the modeling language. The lifting procedure, however, cannot be carried out straight-forwardly, as it has to achieve a shift in focus, which stems from the fact that although metamodeling and ontology engineering share a common ground in conceptual modeling in general, since ontologies and metamodels are designed with different goals in mind. Metamodels prove to be more implementation-oriented as they often bear design decisions that allow producing sound, object-oriented implementations. Due to this, language concepts can be hidden in a metamodel, which during the lifting procedure have to be made explicit in an ontology.

The main contribution of this paper is to lay out the lifting procedure and discuss issues that have to be considered when lifting metamodels to ontologies. Hence, the remainder of this paper is structured as follows: The next section gives a conceptual overview of that lifting process and establishes a big picture in context with the ModelCVS project. Section 3 elaborates on the part of lifting, which deals with a formalism change concerning the way metamodels and ontologies are expressed. Section 4 introduces a pattern catalogue that helps to explicate hidden language concepts and exemplifies its usage. Based on these examples, Section 5 finally shows how the lifting procedure can benefit typical integration tasks such as schema matching. Section 7 discusses related work and Section 8 concludes with an outlook on future work.

2 Lifting at a Glance

A key focus of the ModelCVS project is to provide a framework for semi-automatic generation of transformation programs. Although ModelCVS' architecture allows for an immediate integration of metamodels via specific metamodel integration operators called *bridgings,* of which executable model transformations can be derived, our approach sees a conceptual integration of metamodels via the creation of ontologies from these metamodels as a prerequisite to enhance automation support. As the lifting process results in ontologies explicitly representing the concepts of a modeling language, we propose that matching these ontologies can provide better results in terms of more concise mappings, which in turn can be derived into *bridgings* between the original metamodels. The left-hand side of Fig. 1 shows the general setup of

ModelCVS' architecture, whereas details on the right hand side especially depicting the lifting process will be given throughout the following paragraphs. For more details on ModelCVS we refer the reader to [15],[16].

When trying to lift metamodels to ontologies, the gap between the implementation oriented focus of metamodels and the knowledge representation focus of ontologies has to be closed. Our approach separates the lifting process into three steps. The first step, which we refer to as *conversion*, involves a change of formalism (1), meaning that a metamodel is transformed into an ontology. The transformation is given by a mapping between the model engineering space and the ontology engineering space, namely a mapping from a meta-metamodel (M3) to an ontology metamodel (M2). This transformation results in what we call a *pseudo-ontology*, as the structure of this ontology basically resembles the original metamodel and typically does not represent concepts as explicitly as ontology engineering principles would advise to do.

Hence, in the subsequent *refactoring* step (2), patterns (cf. Section 4) are applied to the resulting pseudo-ontology, which aim at unfolding typically hidden concepts in metamodels that should better be represented as explicit concepts in an ontology. As to be shown in Section 4, however, the decision of *which* pattern should be applied *where,* incorporates new semantics into the model, that were previously retained as part of the user's expert knowledge about the modeling language, only.

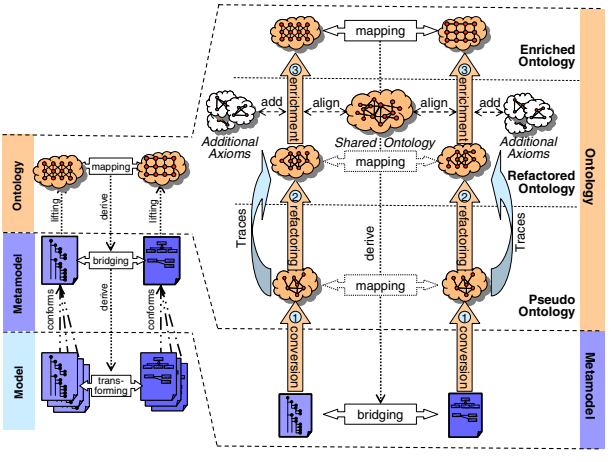

Fig. 1. ModelCVS conceptual architecture

Finally, ontologies being extracted from modeling languages' metamodels can be *enriched* with axioms (3) and put in relation with other ontologies representing a shared vocabulary about a certain domain. Thus, semantic enrichment refers to incorporating *additional* information into ontologies for integration purposes.

Instead of the original metamodels, the resulting ontologies are the driving artifacts that enable *semantic integration* of the associated modeling languages. In our case, we use matching techniques that yield a mapping between two ontologies, which is then the basis for a code generation process that derives model transformations defined between the original metamodels. To be able to relate ontology mappings back

to the original metamodels, traces linking metamodel and ontology constructs have to be established during the lifting process and maintained during the refactoring step. However, a discussion about how our prototype implements the tracing and the code generation mechanisms is considered out of scope of this paper, as is the not obligatory enrichment step. But nevertheless these concepts are necessary to be mentioned to understand the lifting as a part of a meaningful whole and as a prerequisite for operationalizing the discovered mappings in the form of executable model transformations.

3 Conversion - Mapping Ecore to ODM

This section elaborates on a mapping from the model engineering to the ontology engineering technical space. In particular, we focus on describing a mapping from Ecore, which is the meta-metamodel used in the Eclipse Modeling Framework (EMF) [6] that also constitutes ModelCVS' technological backbone, to the Ontology Definition Metamodel (ODM) [12]. This mapping constitutes the basis of our approach, as a transformation based on this mapping is the first step in our lifting process. However, this mapping is not yet introducing any kind of additional semantics into the metamodel and solely provides a change of formalism.

It is relatively easy to find semantic correspondences between Ecore and ODM, as both formalisms are per se fit for conceptual modeling. The goals aimed at when using either formalism, however, differ. Often the intentions behind using a certain construct overlap, like when defining a common superclass for two subclasses to denote that all instances of the subclasses are also instances of the superclass. This intention would be equally satisfied in both Ecore and ODM. However, in Ecore this also means that instances of either subclass can be instance of one of the subclasses only, whereas individuals in OWL could actually belong to both subclasses. These subtle semantic nuances have to be considered when committing to a mapping. Although the definition of a standard metamodel for ontology definition is still under way, the given mapping description refers to terminology used in the latest submission to the ODM RFP [12]. This mapping is similar to a mapping proposition of UML to OWL [12] that can give more details on the partly mechanic part of mapping modeling language constructs to ontology constructs. The next two sub-sections focus on the caveats and the implementation of the Ecore to ODM mapping.

3.1 Caveats of Mapping

The conversion step can ignore meta-classes that do not represent concepts of the modeling language and therefore, should not be lifted into an ontology. In case of Ecore, the classes *EFactory*, *EOperations*, and *EParameter* fall into this category, because these meta-constructs are necessary when generating Java implementation classes from the metamodel, only. Furthermore, the Ecore metamodel contains abstract classes which do not directly take part in the mapping as well, but their concrete subclasses do. Table 1 gives an overview of relevant meta-classes and a catalogue with the appropriate mapping definitions towards the ODM metamodel.

Table 1. Overview of ECore to ODM mapping

Ecore Concept	OWL Concept	Possible Caveat
EFactory, EOperation, EParameter	*no mapping*	*ignored*
EPackage	OWLOntology	inverse hierarchy
EClass	OWLClass	non-exclusive instanceof
EAttribute	OWLDatatypeProperty	name clash / qualification
EReference	OWLObjectProperty	name clash / qualification
EDatatype	RDFSDatatype	*straight-forward*
EEnum & EENumLiteral	OWLDataRange & RDFSLiteral	*straight-forward*
EAnnotation	RDFSLiteral	*straight-forward*

EPackage to OWLOntology. Being both containers for other metaclasses, at first sight, the constructs *EPackage* and *OWLOntology* seem like a straight-forward match. EPackage can be compared to traditional packaging mechanisms as known from other modeling languages, that serves to group and compartmentalize modeling elements or source code. Similarly an *OWLOntology* consists of a collection of ontology elements like cases, properties, axioms and the like. However, the notion of the *eSubpackage* reference cannot be straight-forwardly translated into the *OWLimports* property: An ontology imports another ontology to make use of all the concepts defined in the import. Thus, the top-level ontology has visibility over all imported concepts. Packages on the other hand can have sub-packages, which have visibility over all their super-packages. Hence, the semantics of *subPackage* and *OWLimports* oppose each other. Furthermore, the grouping of model elements in sub-packages lies in the hands of the modeler and basically allows for arbitrary grouping to keep large models comprehensible. The import structure of ontologies is rather based on enabling efficient reasoning and creating a meaningful whole out of certain domain concepts.

Albeit the above mentioned issue, from a pragmatic point of view in most cases it is reasonable to map packages directly to ontologies. Analogously, matching the *subPackage* reference to the *OWLimports* property generally works well, too, when being aware that the result can be an 'up-side down' class hierarchy.

EClass to OWLClass. The metaclasses *EClass* and *OWLClass* map straight-forwardly to an *OWLClass*, except that an *OWLClass* is used to cluster a number of individuals, which can also be individuals of other classes, whereas instances of an *EClass* cannot. This issue, however, does not pose a problem when mapping from Ecore to ODM or when instances are not considered in the lifting process.

The lifting of abstract classes or interfaces depends on whether they represent semantics of the modeling language which should also be represented as concepts in the ontology, or whether they serve solely implementation specific purposes. Our approach follows a strategy of lifting all abstract classes and interfaces, as unnecessarily lifted concepts can usually be better filtered out in the subsequent refactoring step.

EAttribute to OWLDatatypeProperty. In difference to an *EAttribute* belonging to an *EClass*, a property in an ontology is independent of a certain *OWLClass*. Thus, the straight-forward mapping from *EAttribute* to *OWLDatatypeProperty* can be

problematic, because seemingly identical attributes in different classes can carry different semantics, which would then be unified in a single ontology property.

To avoid this problem, one can incorporate additional information like the owning class' name into the name of the newly created property. In doing so, no information gets lost and redundant properties can be joined in the subsequent refactoring step.

EReference to OWLObjectProperty. Similar to the previous mapping description, an *EReference* can be mapped onto an *OWLObjectProperty* when the mentioned name clash problem is dealt with accordingly and the associated loss of semantics is avoided. Apart from this, the *eReferenceType* reference can be mapped to the *RDFSDomain* reference and the *eContainingClass* reference to the *RDFSRange* reference. Just like the former mapping, cardinalities do not pose a problem, as the Ecore references in question have single cardinality which maps straight onto the multiple cardinality of the equivalent references in the ODM.

Summarizing the above remarks, it has to be pointed out that the most important point when defining a mapping from metamodels to ontologies is, that one has to be aware how the resulting ontology is affected by the mapping decisions taken.

3.2 Creating Transformation Code for the Conversion Step

The executable model transformation code facilitating the *conversion* step is created automatically from a mapping specification between Ecore and ODM by means of a code generator. The mapping specification is created with the Atlas Model Weaver (AMW) [7] which is an Eclipse plug-in allowing to weave links between metamodels or models, resulting in a so called weaving model.

In the context of *ModelCVS*, which builds on AMW's weaving mechanism, we more specifically refer to a weaving model as a *bridging*, as it constitutes a mapping specification according to a certain integration scenario [15] of which executable model transformation code can be generated. For defining the mapping between Ecore and ODM we employ a bridging language that denotes a translation of Ecore models into ODM models in a semantics preserving way. This language is defined analogously to a weaving metamodel for the AMW. The semantics of this bridging language is then operationally specified in an adjacent code generator, which produces ATL [14] code that finally performs the actual *conversion* step.

Since the detailed semantics of the bridging language and the inner works of the code generation mechanism are out of scope of this paper and we remain with a general description of the method. In the following paragraphs a rationale for implementing a custom version of the ODM is given.

Since the standardization process for the ODM is still ongoing, a decision was made to implement a custom version of ODM. Our decision was driven by the fact that on one hand, a working import/export functionality of XML serialized OWL ontologies was needed, and on the other hand, an implementation providing an API which reasoners and other ontological software infrastructure could readily use was required. Hence, a decision was made to employ the *Jena* [13] framework that could satisfy both requirements. To be able to bridge the Jena APIs into the model engineering technical space, an Ecore model was reengineered from the Jena API that in the following is referred to as the *Jena ODM*. Wrapping the Jena ODM directly onto the

structure of the underlying API has the advantage, that the writing of an adapter program calling the Jena API to instantiate a Java in-memory model from a Jena ODM model and vice versa boils down to a trivial task. Nevertheless, once a standard is finalized, the described approach can be modified with reasonable effort by defining a transformation from the adopted ODM to the Jena ODM. In MDA terminology, this approach could be compared to a PIM to PSM transformation introducing a new layer of abstraction that helps to keep the adapter program free of transformation logic. For reasons of brevity, we will not further elaborate on implementation details of the conversion step. The output of this first step is a pseudo-ontology, which is the input for the *refactoring* step whose associated patterns will be focused on next.

4 Refactoring Patterns for Pseudo-ontologies

The aim of metamodeling lies primarily in defining modeling languages in an object-oriented manner leading to efficient repository implementations. This means that in a metamodel not necessarily all modeling concepts are represented as *first-class citizens*. Instead, the concepts are frequently hidden in *attributes* or in *association ends*. We call this phenomenon *concept hiding*. Consequently, also *pseudo-ontologies*, i.e., the output of the previous *conversion* step, also lack the explicit representation of modeling concepts. In order to overcome this problem, we propose *refactoring* as a second step in the lifting process, which semi-automatically generates an additional and semantically enriched view of the conversion step's output.

As an example for concept hiding in metamodels consider Fig. 2. In the upper part it shows a simplified version of the *UML metamodel kernel* which is defined in the *UML Infrastructure* [19], represented as a *pseudo*-ontology. As we see in Fig. 2 the *pseudo*-ontology covers *twelve modeling concepts* but uses only *four classes*. Hence, most of the modeling concepts are implicitly defined, only.

To tackle the *concept hiding* problem, we propose certain refactoring patterns for identifying *where* possible hiding places for concepts in metamodels are and also *how* these structures can be rearranged to explicit knowledge representations. The refactoring patterns given in the following subsections are classified into four categories. The description of each pattern is based on [11] and consists of pattern name, problem description, solution mechanism, and finally, of an example based on the UML kernel. The kernel is shown in the upper part of Fig. 2 as a pseudo-ontology (before applying the patterns) and in the lower part of Fig. 2 as a refactored ontology (after applying the patterns). The numbers in the figure identify where a certain pattern can be applied and how that structure will be refactored, respectively.

4.1 Patterns for Reification of Concepts

a) Association Class Introduction: A modeling concept might not be directly represented by object properties but rather hidden within an association. In particular, it might be represented by the combination of both properties representing the context in which these object properties occur.

Refactoring: A new class is introduced in the ontology similar to an association class in UML to explicitly describe the hidden concept. Since there is no language

construct for association classes in OWL, the association is split up into two parts which are linked by the introduced class. The cardinalities of the new association ends are fixed to one and the previously existing association ends remain unchanged.

Example: The combination of the roles of the recursive relationship of *Class*, *subclass* and *superclass*, occurs in the context *generalization*.

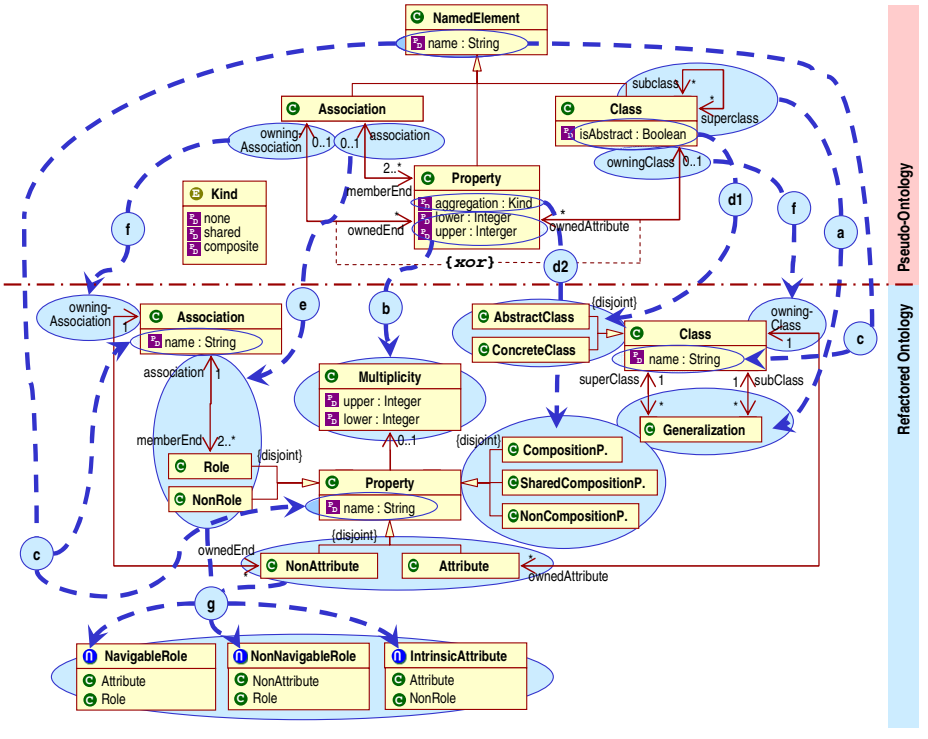

Fig. 2. Part of the UML kernel as pseudo-ontology and as refactored-ontology

b) Concept Elicitation from Properties: In metamodels it is often sufficient to implement modeling concepts as attributes of primitive data types, because the primary aim is to be able to represent models as data in repositories. This approach is in contradiction with ontology engineering which focuses on knowledge representation and not on how concepts are representable as data.

Refactoring: Datatype properties which actually represent concepts are extracted into separate classes. These classes are connected by an object property to the source class and the cardinality of that object property is set to the cardinality of the original datatype property. The introduced classes are extended by a datatype property for covering the value of the original datatype property.

Example: The properties *Property.lower* and *Property.upper* represent the concept *Multiplicity* which is used for defining cardinality constraints on a *Property*.

4.2 Patterns for Elimination of Abstract Concepts

c) Abstract Class Elimination: In metamodeling, generalization and abstract classes are used as a means to gain smart object-oriented language definitions. However, this benefit is traded against additional indirection layers and it is well-known that the use of inheritance does not solely entail advantages. Furthermore, in metamodels, the use of abstract classes which do not represent modeling concepts is quite common. In such cases generalization is applied for *implementation inheritance* and not for *specialization inheritance*. However, one consequence of this procedure is a fragmentation of knowledge about the concrete modeling concepts.

Refactoring: In order to defragment the knowledge of modeling constructs, the datatype properties and object properties of abstract classes are moved downwards to their concrete subclasses. This refactoring pattern yields multiple definitions of properties and might be seen as an *anti*-pattern of object-oriented modeling practice. However, the properties can be redefined with more expressive names (e.g. *hyponyms*) in their subclasses.

Example: The property *NamedElement.name* is used for class name, attribute name, association name and role name.

4.3 Patterns for Explicit Specialization of Concepts

d) Datatype Property Elimination: In metamodeling it is convenient to represent similar modeling concepts with a single class and use attribute values to identify the particular concept represented by an instance of that class. This metamodeling practice keeps the number of classes in metamodels low by hiding multiple concepts in a single class. These concepts are equal in terms of owned attributes and associations but differ in their intended semantic meaning. For this purpose, attributes of arbitrary data types can be utilized but in particular two widespread refinement patterns are through *booleans* and *enumerations*.

d1) Refactoring for Boolean Elimination: Concepts hidden in boolean attribute are unfolded by introducing two new subclasses of the class owning the boolean, and defining the subclasses as disjoint due to the duality of the boolean data type range. The subclasses might be named in an *x* and non-*x* manner but descriptive names should be introduced into the ontology by the user.

Example: Class.isAbstract is either true or false, representing an abstract or a concrete class, respectively.

d2) Refactoring for Enumeration Elimination: Implicit concepts hidden in an enumeration of literals are unfolded by introducing a separate class for each literal. The introduced classes are subclasses of the class owning the attribute of type enumeration and are defined as disjoint, if the cardinality of the datatype property is one, or overlapping if the cardinality is not restricted.

Examples: Property.aggregation is either *none*, *shared*, or *composite*, representing a *nonCompositionProperty,* a *sharedCompositionProperty* or a *CompositionProperty.*

e) Zero-or-one Object Property Differentiation: In a metamodel the reification of a concept is often determined by the occurrence of a certain relationship on the instance

layer. In such cases, the association end in the metamodel has a multiplicity of *zero-or-one* which implicitly contains a concept refinement.

Refactoring: Two subclasses of the class owning the object property with cardinality of zero-or-one are introduced. The subclass which represents the concept that realizes the relationship on the instance layer receives the object property from its superclass while the other subclass does not receive the object property under consideration. Furthermore, the object property of the original class is deleted and the cardinality of the shifted object property is restricted to exactly one.

Example: Property.association has a multiplicity of zero-or-one, distinguishing between a *role* and a *nonRole*, respectively.

f) Xor-Association Differentiation: *Xor*-constraints between *n* associations (we call such associations *xor-associations*) with association ends of multiplicity *zero-or-one* restrict models such that only one of the *n* possible links is allowed to occur on the instance layer. This pattern can be used to refine concepts with *n* sub-concepts in a similar way like enumeration attributes are used to distinguish between n sub-concepts. Thus, *xor*-associations bind a lot of implicit semantics, namely *n* mutually excluding sub-concepts which should be explicitly expressed in ontologies.

Refactoring: This pattern is resolvable similar to the enumeration pattern by introducing *n* new subclasses, but in addition the subclasses are responsible for taking care of the *xor*-constraint. This means each class receives one out of the *n* object properties, thus each subclass represents exactly one sub-concept. Hence, the cardinality of each object property is fixed from zero-to-one to exactly one.

Example: Property.owningAssociation and *Property.owingClass* are both object properties with cardinality zero-or-one. At the instance layer it is determined if an instance of the class *Property* is representing an *attribute* (contained by a class) or a *nonAttribute* (contained by an association).

4.4 Patterns for Exploring Combinations of Refactored Concepts

Refactorings that introduce additional subclasses, i.e., patterns from category Specialization of Concepts, must always adopt a class from the original ontology as starting point since the basic assumption is that different concept specializations are independent of each other. Hence, in the case of multiple refactorings of one particular class, subclasses introduced by different refactorings are overlapping. In Fig. 2 this is denoted using a separate *generalization set* for each refactoring. However, this approach requires an additional refactoring pattern for discovering possible relationships between combinations of sub-concepts.

g) Concept Recombination: In order to identify concepts which are hidden in the ontology as mentioned above, the user has to extend the ontology by complex classes which describe the concepts resulting from possible sub-concept combinations.

Refactoring: User interactions are required for identifying the concepts behind the combination of concepts by evaluating the combinations in a matrix where the dimensions of the matrix are the overlapping generalization sets in consideration.

Example: When studying the textual descriptions of the semantics of UML one finds out that some relationships between the different kinds of *properties* define additional

concepts which are not explicitly represented in the ontology. In particular, the evaluation of *role/nonRole* and *attribute/nonAttribute* combinations leads to the additional intersection classes depicted in the lower part of Fig. 2.

Summarizing, the result of the *refactoring* step, an ontology which facilitates an implementation neutral view of the metamodel, is characterized as follows:

- Only datatype properties which represent semantics of the real world domain (*ontological properties*) are contained, e.g. *Class.className*, *Multiplicity.upper*. This means no datatype properties for the reification of modeling constructs (*linguistic properties*) are part of the refactored ontology.
- Most object properties have cardinalities different from zero-or-one, such that no concepts are hidden in object properties.
- Excessive use of classes and *is-a* relations turns the ontology into a taxonomy.

5 Evaluation of Matching Potential

This section discusses the effects of the refactoring step as defined in the previous section on ontology matching, which is an important task in semantic integration. In particular, we first point out problems in matching pseudo-ontologies that negatively affect matching quality. Subsequently we show how the application of our refactoring patterns can alleviate matching problems and improve mapping quality.

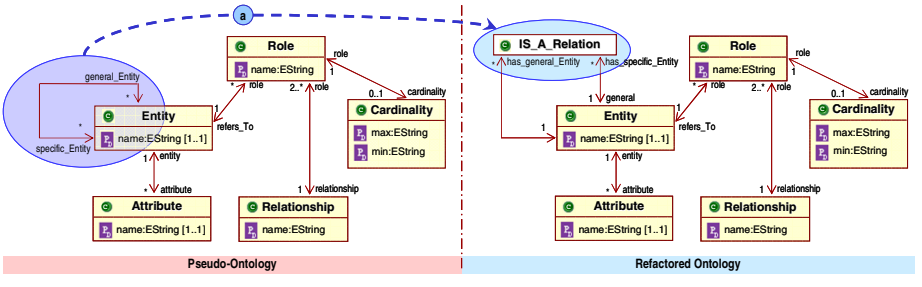

Fig. 3. ER pseudo-ontology (left) and refactored ontology (right)

In our example we are using pseudo-ontologies and refactored ontologies originating from ER and UML metamodels, respectively. The UML ontologies have already been introduced in the previous section, the ER ontologies are depicted in Fig. 3. The ontologies are mapped with COMA++ [2], which allows matching OWL ontologies and produces mappings which represent suggested semantic correspondences. A mapping consists of triples of source element, target element, and a specific confidence rate ranging from zero to one. It is configurable, by associating weights with certain matching rules that can be modified to fit the user's preferences. Hence, the use of COMA++ is naturally a semi-automatic task involving tweaking of the matching algorithm and manual editing of the proposed mapping.

In the following we discuss four general problem classes that can be identified when defining mappings between pseudo-ontologies, and how they become obsolete by applying refactoring. The manifestation of the mapping problems in the UML to

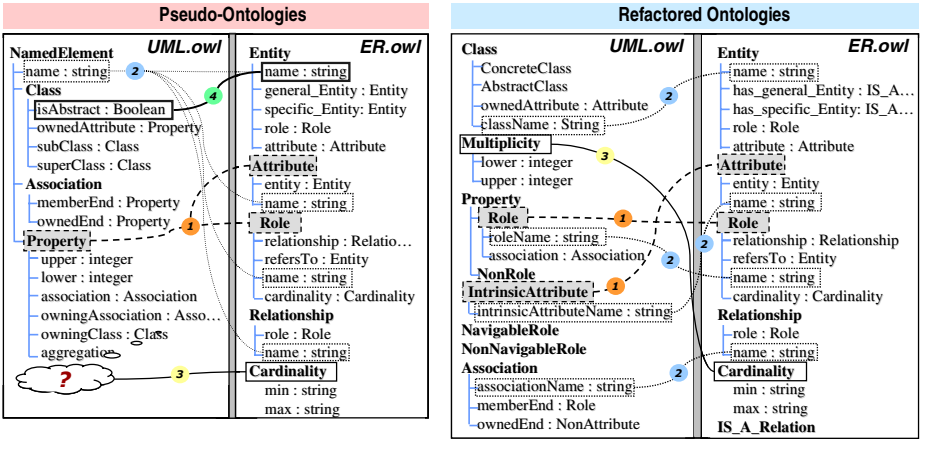

Fig. 4. COMA++ mapping between pseudo-ontologies and refactored ontologies

ER mapping and their solutions using refactored ontologies are shown in Fig. 4. The numbers in that figure refer to the following list of problems:

(1) Ambiguous Concept Mappings: This problem originates from classes in a pseudo-ontology that represent multiple concepts. The example illustrated in Fig. 4 (left) is the mapping from *Property* in UML to *Role* and *Attribute* in ER. This ambiguity arises because the *UML pseudo-ontology* defines a general concept (*Property*) without explicitly stating the sub-concepts which in contrast are represented as explicit concepts in the *ER pseudo-ontology*. This kind of problem is solved by the patterns from the Specialization and the Combination categories, which introduce the hidden concepts as subclasses and complex classes, respectively, thus avoiding ambiguous mappings. In Fig. 4 (right) one can see that the classes introduced from class *Property* allow semantically unambiguous mappings for *roles*, and attributes in the sense of UML *IntrinsicAttribute*.

(2) Ambiguous Property Mappings: The use of abstract classes in a metamodel is a design decision. Hence, when mapping properties that are defined in abstract classes, they may be fragmented over different inheritance layers. This problem is depicted in Fig. 4 (left) by mapping the datatype property *NamedElement.name* to multiple targets. After applying patterns from the Elimination category, the inheritance layers become flattened and the properties are shifted to the subclasses of the abstract classes, thus enabling unambiguous one-to-one mappings. E.g., in Fig. 4 (right) the datatype property *name* of the class *NamedElement* is flatted into the subclasses which lead to unambiguous mappings for the datatype property *name*.

(3) No Counterparts: Pseudo-ontologies might differ in their granularity of modeling concept definitions, although the same modeling concepts are useable by the modeler. Consequently, some mappings cannot be expressed, because explicit concepts of some pseudo-ontology are missing as explicit concept representations in the other. In our mapping example shown in Fig. 4 (left) no corresponding concept in the *UML pseudo-ontology* exists for the *Cardinality* concept of the *ER pseudo-ontology*.

Patterns from the Reification category tackle this problem by the reification of hidden concepts, allowing to define mappings that were not possible before the refactoring step. Concerning the missing counterpart for the *Cardinality* concept, after applying the patterns it is possible to map the *Cardinality* concept to the introduced *Multiplicity* concept as shown in Fig. 4 (right).

(4) Linguistic-to-Ontology Property Mappings: Concerning invalid mappings, one source of defect is mapping linguistic properties to ontological properties. For instance, in our example shown in Fig. 4 (left) *Class.isAbstract* which represents a linguistic property was automatically mapped by COMA++ to *Entity.name* which represents an ontological property. Patterns from the Specialization category transform linguistic properties to concepts, thus tackling this problem, because only ontological properties remain in the refactored ontology. In Fig. 4 (right) one can see that no mappings between linguistic and ontological properties are possible.

When considering the effect of the refactoring step on the mapping process, one can see a higher potential for manually fine-tuning the mapping due to the finer granularity of a refactored ontology. The improvement in mapping potential, however, comes at the cost of performing the refactoring step and of dealing with a higher number of classes. The alternative would be to use a more sophisticated mapping language to describe unambiguous mappings. In contrast, our approach of using refactoring patterns offers a way to solve the discussed mapping problems through simple semantic correspondences, only. Consequently, the overall complexity of the mapping process is decreased due to its splitting into a refactoring part, which brings the pseudo-ontologies to a common granularity and a mapping part, which relies on simple equality mappings that can be generated semi-automatically.

6 Related Work

Our work is to a good deal influenced by efforts which try to close the gap between the model engineering technical space and the ontology engineering technical space. Among these are, e.g. Bezivin et al. [3] who argue for a unified M3 infrastructure and Atkinson [1] who showed that there are plenty of similarities between the two technical spaces and that differences are mostly community-based or of historic nature. Naturally, an M3 unified infrastructure could possibly ease the proposed lifting procedure. Concrete efforts aiming to provide an adequate bridge encompass [8], specifying a mapping from UML to DAML-OIL, and most prominently the submissions to the OMG's ODM RFP [12] also suggesting a mapping from UML to OWL. Although these efforts influenced the mapping proposed in our conversion step, our focus is not on making a rich language like UML fit for ontology modeling, but on extracting meaningful ontologies from metamodels defining modeling languages.

Many other efforts aiming at semantic integration of data also use a procedure that *lifts metadata to ontologies.* These efforts use XML Schemata [26],[5],[10],[24] which are mapped to RDFS or to OWL [9], respectively. [20] carries out an additional normalization step after lifting, but focuses on ameliorating lexical and simple structural heterogeneities, only. All of these approaches are not immediately reusable in our metamodel-centric context, however, and none of the above approaches relies on

refactoring patterns that would allow to make hidden concepts explicit. As an example, [22] lifts XML schemata and states that the resulting ontologies "will be ad-hoc". Our refactoring approach of pseudo-ontologies tries to deal with this problem. Furthermore, the refactored OWL ontologies can be matched without the need for a complex mapping or query language, which addresses the problem identified in [18] that calls for an OWL query language. There is few related work in terms of refactoring ontologies that were created from an underlying metadata representation aiming at a shift in focus as we do. [21] tries to find implicit semantics through linguistic and structural analysis in labels of hierarchical structures on the Web, but seems not applicable to find hidden concepts in modeling languages, nor does it provide means like to reify these. An interesting approach to ontology refactoring is discussed in [4], which, as opposed to our approach, has the goal of pruning an ontology and deriving a schema thereof, that is then refactored towards an implementation oriented focus.

[25] identifies variability, which is the ability to express semantically equal concepts differently, as the reason for different conceptual models being able to meet the same requirements. Our work can be seen as addressing the problems of heterogeneities introduced due to variability, as the refactoring step can help to make concepts explicit in a uniform way, even though they are initially hidden in different ways.

7 Conclusion

In this paper we have introduced the lifting procedure, which allows to create ontologies from metamodels representing modeling languages. The application of refactoring patterns on the resulting ontologies can make originally hidden concepts explicit and thus improve automation support for semantic integration tasks. Although it is not foreseeable that such tasks will ever be fully automated, we believe that support for the at least semi-automatic integration of modeling tools via their modeling languages is feasible. It is easy to see, that such tool integration tasks require proper tool support and methods guiding the integration process themselves.

Lifting metamodels to ontologies is only one important step in realizing the ModelCVS project. Future work will focus on defining specific domain ontologies that can be relied on in the enrichment step to further enhance ontology matching, as well as enhancing the tracing and the code generation mechanisms to automatically derive model transformation programs from higher-level integration specifications.

References

1. Atkinson C.: On the Unification of MDA and Web-based Knowledge Representation Technologies. 1st International Workshop on the Model-Driven Semantic Web (2004)
2. Aumueller, D.; Do, H., Massmann, S.; Rahm, E.: Schema and ontology matching with COMA++. SIGMOD Conference, June, (2005)
3. Bézivin J. et. al.: An M3-Neutral infrastructure for bridging model engineering and ontology engineering. In: Proc. of the First International Conference on Interoperability of Enterprise Software and Applications. Springer, p. 159-171. (2005)
4. Conesa J.: Ontology-Driven Information Systems: Pruning and Refactoring of Ontologies. Doctoral Syposium of 7th Int. Conf. on the Unified Modeling Language, Lisbonl, (2004)

5. Cruz I. F., Xiao Huiyong, Hsu Feihong.: An Ontology-Based Framework for XML Semantic Integration. Int. Database Engineering and Applications Symposium, 217-226 (2004)
6. Eclipse Tools Project: Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf/
7. Didonet Del Fabro M., Bézivin J., Jouault F., Breton E., Gueltas G.: AMW: a generic model weaver. Proc. of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles, (2005)
8. Falkovych K., Sabou M., Stuckenschmidt H.: UML for the Semantic Web: Transformation-Based Approaches. Knowledge Transformation for the Semantic Web. IOS Press, (2003)
9. Ferdinand M. et al.: Lifting XML Schema to OWL, 4th Int. Conf. on Web Engineering (ICWE), Munich, Germany, July, (2004)
10. Fodor O., Dell'Erba M., Ricci F., Spada A., Werthner H.: Conceptual normalisation of XML data for interoperability in tourism. Proc. of the Workshop on Knowledge Transformation for the Semantic Web, Lyon, France, July, (2002)
11. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, (1997)
12. IBM, Sandpiper Software: Fourth Revised Submission to the OMG RFP ad/2003-03-40, www.omg.org/docs/ad/05-09-08.pdf
13. Jena 2 Ontology API, http://jena.sourceforge.net/ontology/
14. Jouault F., Kurtev I.: Transforming Models with ATL: Proceedings of the Model Transformations in Practice Workshop at MoDELS, Montego Bay, Jamaica (2005)
15. Kappel et. al.: On Models and Ontologies - A Layered Approach for Model-based Tool Integration. Modellierung 2006, Innsbruck, March (2006)
16. Kappel et. al.: Towards A Semantic Infrastructure Supporting Model-based Tool Integration. 1st Int. Workshop on Global integrated Model Management, Shanghai, May, (2006)
17. Kappel G., Kapsammer E., Retschitzegger W.: Integrating XML and Relational Database Systems, in WWW Journal, Kluwer Academic Publishers, June, (2003).
18. Lehti P., Fankhauser P.: XML Data Integration with OWL: Experiences and Challenges. Symposium on Applications and the Internet, p. 160, (2004)
19. OMG: UML 2.0 Infrastructure Final Adopted Specification, formal/05-07-05, (2005)
20. Maedche A., Motik B., Silva N., Volz R.: MAFRA - An Ontology Mapping Framework in the Semantic Web. ECAI Workshop on Knowledge Transformation, Lyon, France, (2002)
21. Magnini B., Serafini L., Speranza M.: Making explicit the Semantics Hidden in Schema Models. Proc. of the Workshop on Human Language Technology for the Semantic Web and Web Services, ISWC, Florida, October, (2003)
22. Moran M., Mocan A.: Towards Translating between XML and WSML. 2nd WSMO Implementation Workshop (WIW), Innsbruck, Austria, June (2005)
23. Noy N.F.: Semantic Integration: A Survey Of Ontology-Based Approaches. SIGMOD Record, Special Issue on Semantic Integration, 33 (4), December, (2004)
24. Roser S.: Ontology-based Model Transformation. Doctoral Symposium of the 8th Int. Conference on Model Driven Engineering Languages and Systems, Jamaica, October, (2005)
25. Verelst J., Du Bois B., Demeyer S.: Using Refactoring Techniques to Exploit Variability in Conceptual Modeling. ERCIM-ESF Workshop, Challenges in Software Evolution, (2005)
26. Volz et al.: OntoLIFT. IST Proj. 2001-33052 WonderWeb, Del. 11, (2003)
27. Wimmer M., Kramler G.: Bridging Grammarware and Modelware, in Proc. of Satellite Events at the MoDELS 2005 Conference, Montego Bay, Jamaica, October, (2005)