Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1581–1591
DOI: 10.15439/2016F250
ACSIS, Vol. 8. ISSN 2300-5963

# Towards OntoUML for Software Engineering: Transformation of Rigid Sortal Types into Relational Databases

Zdeněk Rybola, Robert Pergl
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 16000 Praha 6
Email: {zdenek.rybola, robert.pergl}@fit.cvut.cz

***Abstract*—OntoUML is an ontologically well-founded conceptual modelling language that distinguishes various types of classifiers and relations providing precise meaning to the modelled entities. Efforts arise to incorporate OntoUML into the Model-Driven Development approach as a conceptual modelling language for the PIM of application data. In a prequel paper, we have introduced and outlined our approach for a transformation of OntoUML PIM into a PSM of a relational database. In this paper, we discuss the details of various variants of the transformation of Rigid Sortal types of OntoUML.**

## I. Introduction

SOFTWARE engineering is a demanding discipline that deals with complex systems [1]. The goal of software engineering is to ensure high quality software implementation of these complex systems. To achieve this, various software development approaches have been developed.

Model-Driven Development (MDD) is a very popular approach in the recent years. It is a software development approach based on elaborating models and performing their transformations [2]. The product to be developed is described using various types of models specifying the requirements, functions, structure and deployment of the product. These models are used to construct the product using transformations between models and code.

The most usual part of the MDD approach used in the practice is the process of *forward engineering*: transformations of more abstract models into more specific ones. The most common use-case of such process is the development of conceptual data models and their transformation into source codes or database scripts.

To achieve a high-quality software system, high-quality expressive models are necessary to define the requirements for the system [1]. To use such models in the Model-Driven Development approach, the model should define all requirements and all constraints of the system. Moreover, it should hold that more specific models persist the constraints defined in the more abstract models [3].

This research was partially supported by grant by Student Grant Competition No. SGS16/120/OHK3/1T/18.

OntoUML was formulated in 2005 as a graphical modelling language for developing ontologically well-founded conceptual models [3]. As it is based on cognitive science and modal logic, it helps to create expressive models that are able to describe the domain very precisely. As OntoUML is domain-agnostic, it may be used for any domain. In our research, we focus on the domain of software application data and therefore we use OntoUML to create the PIM of the system. Such model can be then transformed into a PSM of the data persistence. However, as OntoUML uses various types of entities and relations to provide additional ontological meaning to the model elements, the transformation needs to deal with these aspects.

As relational databases represent a very common type of data storage, we focus on the transformation of an OntoUML PIM of application data into an ISM of a relational database. To achieve that, we divide the transformation into the following steps:

1) Transform an OntoUML PIM into a UML PIM including all the aspects defined by the OntoUML constructs.
2) Transform the UML PIM with the additional constraints into a PSM of a relational database including the required additional constraints.
3) Transform the PSM with the additional constraints into the ISM to define the constructs in the database to hold the data and maintain the constraints.

In the prequel paper [4], we outlined the various possibilities of the transformation of Sortal universal types used in OntoUML. In this paper, we discuss the details of the transformation of Rigid Sortal types (Kinds and Subkinds) and illustrate various possibilities on examples. The parallel research focused on the transformation of OntoUML Anti-rigid universal types is discussed in the parallel paper [5].

The structure of the paper is as follows: in section II, the work related to our approach including the OntoUML notation is discussed; in section III, the running example of the OntoUML PIM is explained; in section IV, our approach is discussed and illustrated on the running example; in section V, discussion to our approach is provided; finally, in section VI, the conclusion of the paper results is provided.

978-83-60810-90-3/$25.00

## II. Background and Related work

### A. Model-Driven Development

Model-Driven Development (MDD) is a very popular approach in the recent years. It is a software development approach based on elaborating models and performing their transformations [2]. The product to be developed is described using various types of models specifying the requirements, functions, structure and deployment of the product. These models are used to construct the product using transformations between models and code generation.

MDD was originally based on Model-Driven Architecture (MDA) [6] designed by OMG in 2001. MDA defines these types of models:

- Computation Independent Model (CIM),
- Platform Independent Model (PIM),
- Platform Specific Model (PSM),
- Implementation Specific Model (ISM) [7].

Although established already in 2001, there is still deep interest in this approach, as can be seen in recent publications. The book *Model-Driven Software Development: Technology, Engineering, Management* by Stahl et al. [8] provides a great overview of the MDD approach including the terminology, specifications, transformations and case studies. Another book *Model-Driven Software Engineering in Practice* by Brambilla et al. [9] presents the foundations of MDSE approach and also deals with the technical aspects of MDSE including the basics of domain-specific languages, transformations and tools. Also, the survey by da Silva [10] provides a good overview of the MDD approach and terminology related to MDE, MDD and MDA. Another survey was published by Whittle et al. [11] that focused on the support of the MDE approach in tools and provides a taxonomy of tool-related considerations.

The most usual part of the MDD approach used in the practice seems to be the process of *forward engineering*: transformations of more abstract models into more specific ones. The most common use-case of such process is the development of conceptual data models and their transformation into source codes or database scripts. In our research, we focus on the modelling of application data creating a PIM in OntoUML and performing transformations to generate creation scripts of a relational database schema.

### B. UML

Unified Modeling Language (UML) [12], [13] is a popular modelling language for creating and maintaining variety of models using diagrams and additional components [10]. UML defines a set of building blocks – various types of elements (e.g. classes, use cases, components, etc.), relations (e.g. association, generalization, dependency, etc.) and diagrams (e.g. class diagram, use case diagram, sequence diagram, etc.). It defines also the syntax and semantics of models and a general architecture of the model [7]. In context of the data modelling, UML Class Diagram is the notation mostly used to define conceptual models of application data. Also, to describe the structure of a relational database schema, UML Data Model profile as an extension to the UML Class Diagrams may be used [14].

The main elements of a UML Class Diagram are classes, which serve to classify various types of objects in the domain of interest and specify their features and behaviour [13]. Between the classes, associations and generalization/specialization relations can be defined. The associations are used to define the fact that various instances of one class can be related to some instances – according to association multiplicities – of the other class.

Generalization is a taxonomic relationship between a more general class – *superclass* – and a more specific class – *subclass* [13]. It is used in situations when there are multiple special cases of the more general class with additional features and/or specialised meaning. In such a situation, the subclasses inherit all features of their superclass and add their own features, so their instances have all the features of both the superclass and the subclass[1]. As UML is designed following the object-oriented programming approach, an object can be an instance of only one class [7]. As UML is based on object-oriented paradigm, an object is either an instance of the superclass or an instance of the subclass, inheriting the features from the superclass but not being its direct instance.

The subclasses of the same superclass may form a *generalization set* to define a partition of subclasses with common meaning [13]. For each generalization set, two meta-properties should be set to restrict the relation of an instance to the individual subclasses: *isCovering* – expressing whether each instance of the superclass must be also an instance of some subclass in the generalization set – and *isDisjoint* – expressing whether an object can be an instance of multiple subclasses in the set at the same time. The default setting of these properties differ in the various versions of UML: UML 2.4.1 [12] and older define the `{incomplete, disjoint}` as default, while UML 2.5 [13] defines the `{incomplete, overlapping}` as default. As each object is an instance of exactly one class in the most current programming languages, the concept of generalization sets can be used only in conceptual models and it must be transformed before its realization.

### C. OCL

Object Constraint Language (OCL) [15] is a specification language that is part of the UML standard. It can be used for the following purposes:

- to access model elements and their values,
- to define constraints and restrictions for model elements and their values,
- and to define query operations [7].

Several types of OCL constructs may be used to define the constraints for the model elements. *Invariants* are defined in context of certain class of the attached UML model and they are used to define constraints which must be satisfied

[1] In fact, the features of the superclass may be overriden by features of the subclass, but this situation is not considered here.

by all contextual instances at any moment. *Preconditions* and *postconditions* are defined in context of certain method of a class in the attached UML model. Preconditions define the constraints that must be satisfied before executing the method (e.g. the values of the method parameters), while postconditions define the constraints which must be satisfied after the method execution (e.g. the value of the result).

In [16], the authors define basic syntax and semantics of OCL constructs and introduce several tools that support modelling and evaluation of OCL constraints. In [17], the authors define a technique for transformations of OCL constructs into other equivalent forms to support their definition, validation and transformation.

In our approach, we use OCL invariants to define the constraints on the UML PIMs and PSMs derived from the semantics of OntoUML universal types that cannot be expressed directly in the diagram.

### D. OntoUML

OntoUML is a conceptual modelling language focused on building ontologically well-founded models. It was formulated in Guizzardi's PhD Thesis [3] as a light-weight extension of UML based on UML profiles.

The language is based on *Unified Foundational Ontology* (UFO), which is based on the cognitive science and modal logic and related mathematical foundations such as sets and relations. Thanks to this fact, it provides expressive and precise constructs for modellers to capture the domain of interest. Unlike other extensions of UML, OntoUML does not build on the UML's ontologically vague "class" notion, but builds on the notion of *universals* and *individuals*. It uses the basic notation of UML Class Diagram like classes, associations and generalization/specialization together with stereotypes and meta-attributes to define the nature of individual elements more specifically. On the other hand, it omits a set of other problematic concepts (for instance aggregation and composition) and replaces them with its own ontologically correct concepts.

UFO and OntoUML address many problems in conceptual modelling, such as part-whole relations [18] or roles and the counting problem [19]. The language has been successfully applied in different domains such as interoperability for medical protocols in electrophysiology [20] and the evaluation of an ITU-T standard for transport networks [21].

However, being domain-agnostic, we believe that it may be suitable even for conceptual modelling of application data in the context of MDD. Using OntoUML, we can create very precise and expressive models of application data. These models can be later transformed into relational database schema containing various domain-specific constraints to maintain consistency according to the OntoUML model.

The following description of the OntoUML and UFO aspects is based on [3].

*1) Universals and individuals:* UFO distinguishes two types of things. *Universals* are general classifiers of various objects and they are represented as classes in OntoUML (e.g. `Person`). There are various types of universals according to their properties and constraints as discussed later. *Individuals*, on the other hand, are the individual objects instantiating the universals (e.g. `Mark`, `Dan`, `Kate`).

The fact that an individual is an instance of a universal means that – in the given context – we perceive the object *to be* the Universal (e.g. `Mark` is a `Person`). Important feature of UFO is the fact that an individual may instantiate multiple universals at the same time but all the universals must have a common ancestor providing the identity principle (e.g. `Mark` is a `Person` and he is a `Student` as well).

*2) Identity principle:* Identity principle is a key feature of UFO, which enables individuals to be distinguished from each other. Various universals define different identity principles and thus different ways how to distinguish their individuals (e.g. a `Person` is something else than a `University`); different individuals of the same universal have different identities (e.g. `Mark` is not `Kate` even when both are `Persons`).

Each individual always needs to have a single specific identity, otherwise there is a clash of identities (e.g. `Mark` is a `Person` and therefore it can never be confused with another concept such as a `University`). The identity of an individual is determined at the time the individual comes to existence and it is immutable – it can never be changed (e.g. `Mark` will always be `Mark` and he will always be a `Person`).

The types of universals that provide the identity principle for their instances are called *Sortal universals* (e.g. `Person`, `Student`). The types of universals not providing the identity principle are called *Non-Sortal universals* (e.g. a `Customer` may be a `Person` or a `Company`). In this paper, we discuss only the transformations of the Sortal types of universals, as they form the basis of models.

*3) Rigidity:* UFO and OntoUML are built on the notion of worlds coming from Modal Logic – various configurations of the individuals in various circumstances and contexts of time and space. *Rigidity* is, then, the meta-property of universals that defines the fact if the extension of the universal (i.e. the set of all instances of the universal) is world invariant [22]. UFO distinguishes rigid, anti-rigid and semi-rigid universals:

- `Rigid universals` are such types of universals whose extension is rigid – instances of the Rigid universals cannot cease to be their instances without ceasing to exist (e.g. `Mark` will always be a `Person`). Certain types of both Sortal and Non-Sortal universals are rigid.
- `Anti-rigid universals` are such types of universals which, in one world, contain an instance in their extension, which is not included in the extension in another world. It means that an individual that is an instance of the Anti-rigid universal in one world may not be an instance of that universal in another world without ceasing to exist (e.g. `Mark` is a `Student` now, but he will not be a `Student` 50 years later). Certain types of both Sortal and Non-Sortal universals are anti-rigid.
- `Semi-rigid universals` are such types of universals that can include both rigid and anti-rigid instances in

their extension. Only Non-Sortal types of universals are semi-rigid.

In this paper, we focus only on the Rigid Sortal types of universals and we discuss the details of the transformation of such universals into the relational databases.

*4) Generalization and Specialization:* In contrast to UML, in UFO and OntoUML, the generalization relation defines the inheritance of the *identity principle*. According to that, an individual which is an instance of the subclass is also an instance of the superclass automatically through inheriting the identity principle from the superclass. Also, the relation is rigid in UML – when an instance of the superclass is also an instance of the subclass, it cannot cease to be so without losing its identity – while in OntoUML, the relation may be non-rigid: a single individual may be an instance of both the superclass and subclass in one world and it may be an instance of only the superclass in another world.

The generalization sets in OntoUML are much more common as they define the required identity for various universal types. Unless altered, *{incomplete, non-disjoint}* is considered the default value of the meta-properties.

*5) Kinds and Subkinds.:* The backbone of an OntoUML model is created by Kinds. *Kind* is a Rigid Sortal type of universals that defines the identity principle for its instances, thus defining the way how we are able to distinguish individual instances of that universal. In OntoUML, the Kind universals are depicted as classes with the $\ll Kind \gg$ stereotype. Examples of Kind universals are a `Person` and a `University`.

*Subkind* is a Rigid Sortal universal type that does not define its own identity principle, but it inherits it from its ancestor and provides it to its instances. Therefore, Subkind universals form generalization sets of other Kind or Subkind universals; they form inheritance hierarchies with the root in a Kind universal. In other words, each instance of a Subkind universal is automatically – through the transitive generalization relation – also an instance of all the ancestral Kind and Subkind universals, receiving the identity principle from the root Kind universal. The inheritance may have any combination of values of the *isDisjoint* and *isCovering* meta-properties. Examples of Subkind universals may be a `Man` and a `Woman` as subkinds of a `Person`.

*6) Other universal types.:* UFO and OntoUML define several other universal types such as Role, Phase, Relator, Mixin, Quantity et al. However, they are out of scope of this paper.

### E. Tools

There are tools supporting certain parts of the transformation process described in section IV. Although none of them supports the full transformation, they can be used for the individual steps or serve as an inspiration for a complex tool to be developed.

Enterprise Architect[2] is a complex CASE tool supporting the whole software development process. Beside the modelling in UML and other notations, it offers transformation between models and source code generation. In context of our work, the transformation of a class model into a database model and the generation of SQL DDL scripts are useful. Beside Enterprise Architect, there are many other tools providing similar functions for UML and relational databases (e.g. Visual Paradigm[3]).

There are also several tools supporting definition of OCL constraints and their evaluation on a given model instance, such as DresdenOCL[4], OCLE[5] or USE[6]. DresdenOCL even provides functions to generate Java source code with AspectJ for the OCL constraints or SQL DDL scripts with views for the OCL constraints.

For OntoUML, there are a few tools available, as well. OntoUML lightweight editor (OLED)[7] is an environment for modelling with OntoUML which also offers functions for model visualisation, validation and transformation into OWL. However, it does not offer transformation into UML nor into relational databases. Menthor Editor[8] is a successor of OLED, providing more convenient environment for modelling and providing transformations of an OntoUML model along with OCL constraints into OWL, RDF and UML. As for other tools, there is an Enterprise Architect plugin[9] and a palette for UMLet editor[10] available for OntoUML modelling.

### F. Previous work

In our previous work, we focused on the transformation of special multiplicity values in a UML PIM into PSM for relational databases [23] and the possible realizations of such constraints [24]. The approaches described in these papers may be used for the realization of the constraints derived from the OntoUML constructs used in the PIM as discussed in this paper.

In [25] we focused on the transformation of an ontological conceptual model in OntoUML into a pure object implementation model in UML and also the instantiation of such model to validate it. In the paper [4], we outlined our approach to the transformation of OntoUML PIM into an ISM of a relational database. In the parallel paper [5], we discuss the details of the transformation of OntoUML Anti-rigid Sortal types. This paper presents the parallel research focused on the transformation of OntoUML Rigid Sortal types (Kinds and Subkinds).

## III. Running Example

Our approach to the transformation of the Sortal Rigid universal types in an OntoUML PIM into ISM of a relational database is illustrated on the running example shown in Figure 1. The model shows an excerpt of the domain of an automotive company. The company takes care about

[2]http://www.sparxsystems.com.au/products/ea/

[3]https://www.visual-paradigm.com/

[4]https://github.com/dresden-ocl

[5]http://lci.cs.ubbcluj.ro/ocle/

[6]http://sourceforge.net/projects/useocl/

[7]https://github.com/nemo-ufes/ontouml-lightweight-editor

[8]http://www.menthor.net/menthor-editor.html

[9]http://www.menthor.net/ea-plugin.html

[10]https://zenodo.org/record/51859

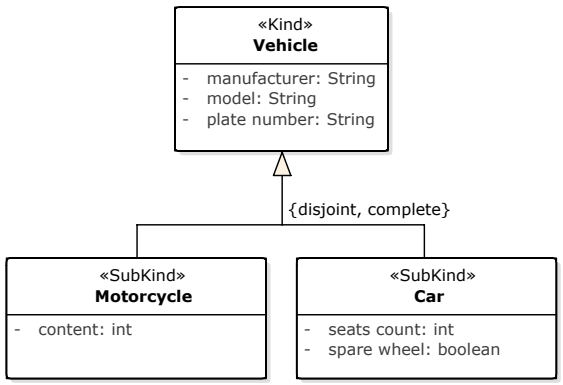


Fig. 1. OntoUML PIM of vehicle types

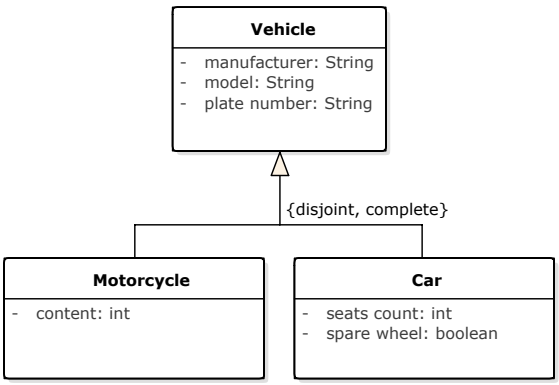


Fig. 2. UML PIM of vehicle types

motorcycles and personal cars in their fleet. These two types of vehicles are represented by the `Motorcycle` and `Car` Subkinds of the common ancestral Kind `Vehicle` defining the identity of a vehicle.

As the company uses only motorcycles and personal cars, the generalization set is `complete`. Also, it is not possible for a single vehicle to be both the motorcycle and the car, therefore the generalization set is `disjoint`.

## IV. Our Approach

Our approach to the transformation of a PIM in OntoUML into its realization in a relational database consists of three steps which are discussed in the following sections:

1) subsection IV-A discusses the transformation of an OntoUML PIM into a UML PIM,
2) subsection IV-B discusses the transformation of the UML PIM into a PSM for relational database,
3) subsection IV-C discusses the transformation of the PSM into an ISM of the relational database.

As mentioned in the introduction, it should hold that no information should be lost when transforming from a more abstract model into a more specific one. As OntoUML applies certain constraints based on the OntoUML type used for an entity, these constraints should be carried over to the other models. In our approach, we use OCL to define such constraints in the UML models that cannot be expressed directly in the diagrams.

Although we may formulate a direct transformation from OntoUML into the relational database, the transformation via an auxiliary UML model enables to leverage all the available knowledge (e.g. [26], [24] and tools for transformation of a UML PIM into database models such as Enterprise Architect[11]. Also, various optimizations and refactoring may be applied whenever possible (e.g. when the entity does not hold any attributes, they can be expressed by a mere attribute of the superclass).

In the approach presented here, we assume the (most common) situation where all attributes of the model classes have multiplicities `[1..1]`. In the conclusions, we discuss how the situation changes for different multiplicities.

[11]http://www.sparxsystems.com.au/products/ea/

### A. Transformation of OntoUML PIM into UML PIM

This phase of the transformation deals with the transformation of various types of universals in an OntoUML model into a pure UML model while preserving all the semantics defined by the universal types.

In this phase of the transformation, the semantics of the OntoUML model is mostly realized by the multiplicities of the relations between the classes in the UML model.

As various OntoUML universal types define different semantics, they are also transformed in a different manner. However, we discuss only the transformation of Rigid Sortal types (Kinds and Subkinds) and their variants in this paper.

*1) Kinds and Subkinds:* As both Kind and Subkind universals in OntoUML are rigid, their instances cannot cease to be their instances without ceasing to exists. The same applies in UML for the relation between the instances and their classes. Therefore, the representation of Kinds and Subkinds in UML may stay the same: each $\ll Kind \gg$ and $\ll Subkind \gg$ class is transformed into a standard UML class keeping all its features – attributes and relations.

The resulting transformed PIM into UML for the vehicle domain shown in Figure 1 is shown in Figure 2. Each of the $\ll Kind \gg$ and $\ll Subkind \gg$ classes has been transformed into standard UML class.

*2) Generalization sets:* A Subkind in OntoUML represents a special case of a Kind or other Subkind, forming a generalization set together with other Subkinds. As both Kinds and Subkinds are rigid, also the generalization set is rigid: when an object is an instance of the Subkind, it is also an instance of its rigid ancestor – a Kind or another Subkind – and it cannot cease to be the instance of any of them without loosing its identity.

Thanks to the rigidity, the generalization sets of $\ll Subkind \gg$ classes in the OntoUML model can be transformed into the standard UML generalization set. Also, the meta-properties `isDisjoint` and `isCovering` of the

generalization set remain the same. The example of this transformation can be seen in Figure 2.

### B. Transformation of PIM into PSM

The second step is the transformation of the UML PIM into a PSM of a relational database. The UML Data Model profile – an extension to the UML class diagrams – is used in the examples to define the structure of relational databases in UML [14]. Additional constraints required to preserve the semantics derived from the OntoUML model are defined as OCL invariants, as OCL is part of the UML standard and there are tools supporting the transformation of OCL constraints into database constructs such as DresdenOCL[12]. The basics of this transformation was already discussed in [24]. In this paper, we focus on the transformation of the constraints derived from the OntoUML Rigid Sortal universal types.

In general, when performing transformation from a UML PIM into a PSM of a relational database, classes are transformed into database tables, class's attributes are transformed into table columns and associations are transformed into FOREIGN KEY constraints. Also, PRIMARY KEY constraints are defined for unique identification of individual rows in the tables.

The transformation of classes representing various Kind universals is straightforward – the class with its attributes is transformed into a table with its columns as discussed in [24]. However, more complicated situation arises for the subclasses representing the Subkind universals, as they always form generalization sets. There are multiple standard variants of the transformation of generalization [27], however, they have certain limitations regarding the OntoUML Subkind universal constraints, as discussed in the following sections.

*1) Single table:* In this variant of generalization realization, the superclass and all its subclasses are realized by a single table. Such table contains the columns for all the attributes of the superclass and all its subclasses. Instances of the superclass are represented by rows with the subclasses' columns containing NULL values, instances of a subclass contain values only in the superclass columns and their respective subclass columns – the other columns remain NULL. Usually, a special column to discriminate the subtypes is also defined in the table. The resulting transformed model of the PIM in Figure 2 is shown in Figure 3, where the column `id` serves as the PRIMARY KEY and the column `type` serves as the discriminator.

As our assumption is that attribute multiplicities are `[1..1]` – as mentioned at the beginning of this section – all columns of a class should be NOT NULL in the table. This can be easily defined by the NOT NULL constraints for the columns of the superclass, as they have values even for the instances of the subclasses. However, the constraints for the subclasses' columns depend on the subclass of the instance, which the row represents – the other columns may contain NULL values. Moreover, all columns of a single subclass – not only a subset of them – should have a value.

[12]https://github.com/dresden-ocl

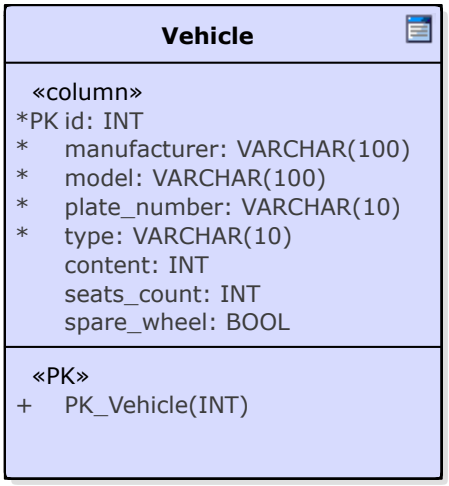


Fig. 3. PSM of vehicle types realized by a single table

As such constraints are not defined on the column level, they cannot be captured directly in the UML model. Instead, they must be defined as additional OCL invariants that are later transformed into their realization in a relational database (see subsection IV-C). For the example in Figure 3, these constraints can be defined for the individual subclasses as shown in Algorithm 1.

Furthermore, the constraints should be defined in respect to the meta-properties of the generalization set according to the following variants:

- `{complete, disjoint}`: For each row, all the columns of the superclass and all the columns of a single subclass must contain NOT NULL values, the other columns must contain NULL values.
- `{complete, overlapping}`: For each row, all the columns of the superclass and all the columns of at least a single subclass cannot contain NULL values.
- `{incomplete, disjoint}`: For each row, all the columns of the superclass must contain NOT NULL values. At the same time, all the columns of at most one of the subclasses may contain NOT NULL values, the other columns must contain NULL values.
- `{incomplete, overlapping}`: For each row, all the columns of the superclass must contain NOT NULL values. All the columns of any subclass may or may not contain NOT NULL values.

Such OCL invariant for the example shown in Figure 3 is shown in Algorithm 2. Because the generalization set is `{disjoint, complete}`, the `Vehicle` columns must contain a value – this is achieved by the NOT NULL constraints of the columns – and the columns of `Motorcycle` class must contain NOT NULL values while the `Car` class must contain NULL values, and vice versa.

The constraints discussed above become the more complicated the more attributes there are in the subclasses because

**Algorithm 1** OCL invariants for the NOT NULL constraints of the subclasses

```
context v:Vehicle inv MotorcycleNotNull:
v.type = 'Motorcycle' implies v.content <> OclVoid

context v:Vehicle inv CarNotNull:
v.type = 'Car' implies v.seats_count <> OclVoid and v.spare_wheel <> OclVoid
```

**Algorithm 2** OCL invariant for the `{disjoint, complete}` generalization set realized by a single table

```
context v:Vehicle inv MotorcycleOrCar:
def: validMotorcycle:Boolean =
        v.content <> OclVoid and v.seats_count = OclVoid and v.spare_wheel = OclVoid
def: validCar:Boolean =
        v.content = OclVoid and v.seats_count <> OclVoid and v.spare_wheel <> OclVoid
validMotorcycle xor validCar
```

of the exclusivity of the NOT NULL values. Therefore, we would recommend this variant of the transformation only in cases there are not many subclasses and their attributes.

*2) Subclasses' tables:* In this variant of the transformation, the tables are created only for the subclasses. The attributes of the superclass are transformed into columns in all the tables of all the subclasses. Therefore, each instance of a subclass is able to store also the values of the superclass's attributes along with their own in a single table. Because of this, all the NOT NULL constraints can be easily defined for all columns.

However, in this variant, it is more complicated to ensure the unique values for attributes of the superclass, as the data are distributed in several distinct tables.

Also, this variant cannot be used for an `incomplete` generalization set as it does not allow the storing of an instance of only the superclass. Even if the NOT NULL constraints on the subclasses' columns would not be defined, it would not be clear in which table to store the instance[13].

Moreover, this variant is not suitable for `overlapping` generalization sets either, as storing the data of multiple subclasses to their respective tables also duplicate the data of the superclass.

Therefore, based on the mentioned restrictions and complications, we would not recommend this variant of the transformation in any situation and we will not discuss it anymore.

*3) Superclass and subclasses' tables:* According to this variant, the superclass and all the subclasses are transformed each into their own table and the individual subclass's tables contain the FOREIGN KEY referring to the superclass's table. This direction is determined by the fact that an instance of the subclass is also an instance of the superclass. Therefore a record in the subclass's table requires exactly one record in the superclass's table – thus being related to `1..1` parent records. More details about determination of the FOREIGN KEY direction based on the multiplicities can be found in [28].

In this variant, the NOT NULL constraints are easier to define, as all columns in a table represent attributes of the same class and they can be expressed by simple NOT NULL constraints defined directly for each column. Still, an additional constraint must be defined for the meta-properties of the generalization set of the subclasses according to the following combinations:

- `{complete, disjoint}`: exactly one row from only one of the subclass's tables refers to the row in the superclass's table.
- `{complete, overlapping}`: at most one row from each of the subclass's tables refers to the row in the superclass's table, but at least one in total.
- `{incomplete, disjoint}`: at most one row from only one of the subclass's tables refers to the row in the superclass's table.
- `{incomplete, overlapping}`: at most one row from each of the subclass's tables refers to the row in the superclass's table.

The restriction of *at most one row from a table* can be realized by a UNIQUE KEY constraint on the FK column; the same column may also be part of the PRIMARY KEY constraint. However, the exclusivity must be checked by a special constraint, still.

In the running example, the subclasses `Motorcycle` and `Car` are transformed into their respective tables (see Figure 4). As their generalization set is `complete` and `disjoint`, the FK column is part of the PRIMARY KEY constraint to make it unique. Furthermore, the constraint shown in Algorithm 3 must be defined.

The other variants of the generalization meta-properties are not discussed here.

[13]Technically, it is possible to designate one of the subclass's tables for this purpose. However, we find this approach inconceptual.

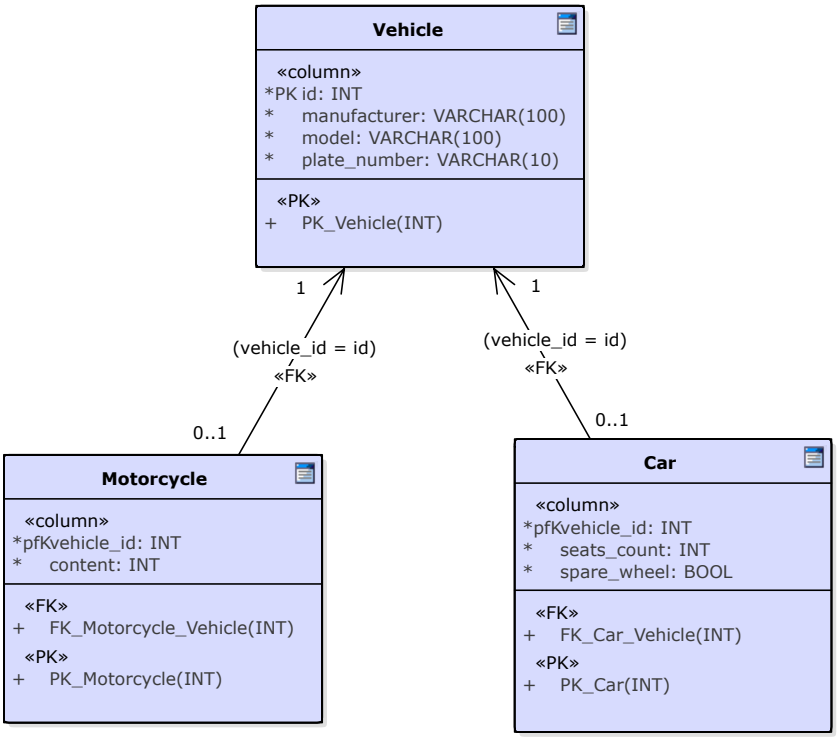


Fig. 4. PSM of vehicle types realized by a separate superclass's and subclesses' tables

**Algorithm 3** OCL invariant for the `{disjoint, complete}` generalization set realized by a superclass's and subclasses' tables

```
context v:Vehicle inv MotorcycleOrCar:
def:validMotorcycle:Boolean = Motorcycle.allInstances()->exists(m|m.vehicle_id=v.id)
def:validCar:Boolean = Car.allInstances()->exists(c|c.vehicle_id = v.id)
validMotorcycle xor validCar
```

### C. Transformation of PSM into ISM

The last step is the transformation of the PSM of a relational database into an ISM. This model consists of database scripts for the creation of the database tables, constraints and other constructs.

As we have the PSM of the relational database, the transformation is quite easy. Most of the current CASE tools such as Enterprise Architect or Visual Paradigm[14] can be used to generate SQL DDL scripts. These scripts usually include the CREATE commands for the tables, their columns, NOT NULL constraints and PRIMARY and FOREIGN KEY constraints.

However, the OCL invariants defined for the additional constraints require special transformation. Only a few tools currently seem to offer transformation of such constraints – e.g. DresdenOCL[15], OCLE[16] or USE[17].

[14]http://www.visual-paradigm.com/
[15]https://github.com/dresden-ocl
[16]http://lci.cs.ubbcluj.ro/ocle/
[17]http://sourceforge.net/projects/useocl/

Our approach to the realization of the OCL constraints derived from the OntoUML universal types is inspired by the approach for special multiplicity constraints discussed in [24]. Based on that approach, the following constructs may be used to prevent violating the derived constraints:

- *Database views* can be used to query only the valid data meeting the constraints. They do not slow down the DML operations, but they do not prevent inserting data violating the constraints.
- *Updatable database views with CHECK option* can be used to manipulate only the valid data, preventing to create invalid data by insert, update and delete operations. However, the use of such views is restricted by several constraints for the query expression.
- *CHECK constraints* can be used to check the values inserted to various columns of the table, but the common current database engines (e.g. Oracle 11g) do not support subqueries in the CHECK constraint expression and therefore they cannot be used for relational constraints.

- *Triggers* can be defined on the DML operations to prevent creating invalid data in the tables. In the triggers, complex queries and checks can be realized, and therefore they are capable to deal with almost every possible constraint. The constraint checks slow down each DML operation, however as shown in [24], the time increase is typically not substantial.

In [24], the research was focused on the realization of special multiplicity constraints. The same approach, however, may be used also for the realization of the constraints derived from the Rigid Sortal universal types and their generalization sets in OntoUML.

In the following sections, the transformation of the resulting PSMs from subsection IV-B is discussed using the approaches listed above (marked by italics).

*1) Single table:* For the generalization set transformed into a single database table, the constraint was defined as shown in Algorithm 2. Its realization using the *database view* is simple: query only such rows from the table that have either the `Motorcycle` columns or the `Car` columns filled with values. The resulting database view is shown in Algorithm 4. The WHERE condition filters out such vehicles that are neither motorcycle nor car as well as such vehicles that are both – and it exactly meets the `{complete, disjoint}` property of the generalization set.

Moreover, as the view definition meets the constraints for an *updatable view*, it can be defined WITH CHECK OPTION and used even for DML operations like inserts, updates and deletes. The WITH CHECK OPTION makes the database engine to check the view after each such operation executed on the view and prevents inserting a row that will not be accessible by the view or updating a row to make it inaccessible.

The same effect might be achieved also by defining a *CHECK constraint* which is checked after each operation on the table. The resulting CHECK constraint is shown in Algorithm 5. Using such constraint, it is not possible to create invalid data in the table and therefore the table can be used directly for querying the valid data.

As the CHECK constraint does not contain any subqueries, it is supported by the common database engines without any problems. The realization by triggers would achieve the same results but with more complex definition and slower evaluation. Therefore, it is not worth to use the *triggers* approach in this case.

*2) Superclass's and subclasses' tables:* For the generalization set transformed into database tables for the superclass and all the subclasses, the constraint was defined as shown in Algorithm 3.

The resulting *database views* for the realization of the OCL invariant are shown in Algorithm 6. The view `Valid_Vehicles` is used to query only such rows from the `Vehicle` table that have a row either in the `Motorcycle` or in the `Car` table referring to it. Therefore, using this view, we can access data about such vehicles that are either a motorcycle or a car, the vehicles having invalid data are hidden from the view.

To query the data of valid motorcycles, the view `Valid_Motorcycles` can be used that filters out invalid motorcycles using the `Valid_Vehicles` view. By analogy, the view `Valid_Cars` can be used to query only the valid cars.

All of these views are updatable – meeting the criteria for an *updatable view* – and therefore they can be defined WITH CHECK OPTION and used to manipulate with the vehicles to prevent creating vehicles without the motorcycle or car data. However, it would not be possible to insert data into any of the views as inserting into the `Valid_Vehicles` would violate the view condition while inserting into the `Valid_Motorcycles` or`Valid_Cars` would violate the FOREIGN KEY constraint. Therefore, the FOREIGN KEY constraint must be defined as *deferrable*, so it is checked at the end of the transaction and not at the time of execution of the command. Then, it is possible to insert data to the `Valid_Motorcycles` or `Valid_Cars` views, first referring to a not-existing vehicle and then to insert data into the `Valid_Vehicles` view.

However, the existence of such views does not prevent the manipulation with the data directly in the tables and thus violating the constraints. Therefore, another database construct should be used. A *CHECK constraint* might have been used as discussed in subsubsection IV-C1, however, as the constraint would contain subquery, it is not supported by the common current database engines [24]. Therefore, the CHECK constraint cannot be actually used in this situation.

Instead, according to the *triggers* approach, triggers might be defined for each of the tables for all the DML operations – insert, update, delete – to check that the operation will not violate the constraint. The following triggers would be needed:

- `BEFORE INSERT ON Vehicle`: This trigger would check there are car data or motorcycle data available in their respective tables for the vehicle. If violated, an error is raised and the operation is cancelled.
- `BEFORE UPDATE OR DELETE ON Motorcycle`: The trigger would check there are no vehicle data in the `Vehicle` table, to which the updated or deleted rows refer. If violated, an error is raised and the operation is cancelled.
- `BEFORE UPDATE OR DELETE ON Car`: The similar trigger should be defined as for the `Motorcycle` table.

Defining such triggers, along with the FOREIGN KEY constraints a PRIMARY KEY constraints, would prevent creating invalid data in the tables during any DML operation. However, the FOREIGN KEY constraints must be defined as *deferrable* – same as for the views – to allow inserting the subclasses' data before inserting the superclass's data or to delete the superclass's data before deleting the subclasses' data.

## V. Discussion

As mentioned above, our approach to the realization of the constraints derived from the OntoUML Sortal universal types is based on the approach discussed in [24]. In this paper, the authors discuss possible ways to realize constraints for

**Algorithm 4** Database view to query valid data from the combined `Vehicle` table

```
CREATE VIEW MotorcycleOrCar AS
SELECT * FROM Vehicle v WHERE
  (v.content IS NOT NULL AND v.seats_count IS NULL AND v.spare_wheel IS NULL)
  OR
  (v.content IS NULL AND v.seats_count IS NOT NULL AND v.spare_wheel IS NOT NULL)
WITH CHECK OPTION
```

**Algorithm 5** CHECK constraint for the combined `Vehicle` table

```
ALTER TABLE Vehicle ADD CONSTRAINT MotorcycleOrCar CHECK
  (v.content IS NOT NULL AND v.seats_count IS NULL AND v.spare_wheel IS NULL)
  OR
  (v.content IS NULL AND v.seats_count IS NOT NULL AND v.spare_wheel IS NOT NULL)
```

**Algorithm 6** Database views to query only valid data from the `Vehicle`, `Motorcycle` and `Car` tables

```
CREATE VIEW Valid_Vehicles AS
SELECT * FROM Vehicle v WHERE
  (EXISTS (SELECT 1 FROM Motorcycle m WHERE m.vehicle_id = v.id)
       AND NOT EXISTS (SELECT 1 FROM Car c WHERE c.vehicle_id = v.id))
  OR
  (NOT EXISTS (SELECT 1 FROM Motorcycle m WHERE m.vehicle_id = v.id)
       AND EXISTS (SELECT 1 FROM Car c WHERE c.vehicle_id = v.id))

CREATE VIEW Valid_Motorcycles AS
SELECT v.*, m.content FROM Valid_Vehicles v
  JOIN Motorcycle m ON (v.id = m.vehicle_id)

CREATE VIEW Valid_Cars AS
SELECT v.*, c.seats_count, c.spare_wheel FROM Valid_Vehicles v
  JOIN Car c ON (v.id = c.vehicle_id)
```

special multiplicity values using database views and triggers. The authors also provide results of experiments, proving that their realization guarantees database consistency in context of the multiplicity constraints with just a slight decrease in efficiency.

The OCL constraints derived from the OntoUML Sortal universal types have the same structure – they are based on multiplicities of related objects or their exclusivity. Therefore, also their realization using the views and triggers is very similar. Based on this, we can expect the same impact on the efficiency of the DML operations and queries. However, as our research is not yet fully concluded, experiments are yet to be done to prove that.

Also, in this paper, we focused on the most common situation of mandatory attributes (attribute multiplicity `[1..1]`). In case of optional attributes (minimal multiplicity `0`), some of the constraints will simplify – e.g. the NOT NULL constraints for individual columns representing the attributes of the subclasses (Algorithm 1 and Algorithm 2). On the other hand, collection attributes (attributes with the maximal multiplicity $*$) lead to the realization in the form of relations, references and FOREIGN KEYs.

## VI. Conclusions

In this paper, we introduced our approach to the transformation of an OntoUML PIM of application data into an ISM of a relational database. This transformation is separated into three sequential steps: the transformation of an OntoUML PIM into a UML PIM, the transformation of the UML PIM into a PSM for relational database and the transformation of the PSM into an ISM of a relational database.

During these transformations, various options and additional constraints should be defined and realized to maintain the semantics defined by the OntoUML universal types. In this paper, we discussed the details of the transformation of Rigid Sortal universal types – Kinds and Subkinds and their generalization sets – discussing various possible realizations of the constraints derived from the semantics of these OntoUML

constructs. All the variants are described using a running example of a simple OntoUML PIM of vehicle types.

As for the future research, a similar work should be elaborated for the Non-sortal universal types – e.g. Category, Mixin, RoleMixin – and relational constructs – part-whole relations, Relators, etc. Also, combinations of multiple generalization sets of a single universal with various combinations of the meta-properties should be investigated. Finally, experiments should be carried out to study the finer points of individual variants of the constraints realization.

## REFERENCES


[1] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002. ISBN 0133056996

[2] S. J. Mellor, A. N. Clark, and T. Futagami, "Model-driven development," *IEEE Software*, vol. 20, no. 5, p. 14, Sep. 2003.

[3] G. Guizzardi, *Ontological Foundations for Structural Conceptual Models*. Enschede: University of Twente, 2005, vol. 015, no. CTIT Ph.D.-thesis series No. 05-74. ISBN 90-75176-81-3

[4] Z. Rybola and R. Pergl, "Towards OntoUML for Software Engineering: Introduction to the Transformation of OntoUML into Relational Databases," in *Enterprise and Organizational Modeling and Simulation*, ser. LNBIP. CAiSE 2016, Ljubljana, Slovenia: Springer, June 2016, in press.

[5] ——, "Towards OntoUML for Software Engineering: Transformation of Anti-Rigid Sortal Types into Relational Databases," in *Model and Data Engineering*, ser. LNCS, vol. 9893. MEDI 2016, Alméria, Spain: Springer, Sep 2016. doi: 10.1007/978-3-319-45547-1_1. ISBN 978-3-319-45546-4. ISSN 0302-9743 pp. 1–15.

[6] OMG, "MDA guide revision 2.0," http://www.omg.org/cgi-bin/doc?ormsc/14-06-01, Jun. 2014, accessed: 2016-03-10.

[7] J. Arlow and I. Neustadt, *UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)*. Addison-Wesley Professional, 2005. ISBN 0321321278

[8] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen, *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013. ISBN 0-470-02570-0

[9] M. Brambilla, J. Cabot, and M. Wimmer, "Model-Driven Software Engineering in Practice," *Synthesis Lectures on Software Engineering*, vol. 1, no. 1, pp. 1–182, Sep. 2012. doi: 10.2200/S00441ED1V01Y201208SWE001

[10] A. R. da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, vol. 43, pp. 139 – 155, 2015. doi: 10.1016/j.cl.2015.06.001

[11] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?" in *Model-Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Sep. 2013, no. 8107, pp. 1–17. ISBN 978-3-642-41532-6, 978-3-642-41533-3. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41533-3_1

[12] OMG, "UML 2.4.1," http://www.omg.org/spec/UML/2.4.1/, Aug. 2011, accessed: 2016-02-08.

[13] ——, "UML 2.5," http://www.omg.org/spec/UML/2.5/, Mar. 2015, accessed: 2016-02-08.

[14] G. Sparks, "Database Modeling in UML," accessed: 2016-02-02. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1255046

[15] OMG, "Object constraint language (OCL), version 2.4," http://www.omg.org/spec/OCL/2.4/, Feb. 2014, accessed: 2016-02-23.

[16] M. Richters and M. Gogolla, "OCL: syntax, semantics, and tools," in *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*. Springer-Verlag, 2002. ISBN 3-540-43169-1 pp. 42–68.

[17] J. Cabot and E. Teniente, "Transformation techniques for OCL constraints," *Science of Computer Programming*, vol. 68, no. 3, pp. 179–195, Oct. 2007. doi: 10.1016/j.scico.2007.05.001

[18] G. Guizzardi, "The problem of transitivity of part-whole relations in conceptual modeling revisited," Amsterdam, The Netherlands, 2009.

[19] ——, "Agent roles, qua individuals and the counting problem," *Software Engineering of Multi-Agent Systems*, no. IV, 2006.

[20] B. Goncalves, G. Guizzardi, and J. G. Pereira Filho, "Using an ECG reference ontology for semantic interoperability of ECG data," *Special Issue on Ontologies for Clinical and Translational Research*, 2011. doi: 10.1016/j.jbi.2010.08.007

[21] P. P. F. Barcelos, G. Guizzardi, A. S. Garcia, and M. Monteiro, "Ontological evaluation of the ITU-T recommendation g.805," vol. 18. Cyprus: IEEE Press, 2011. doi: 10.1109/CTS.2011.5898926

[22] G. Guizzardi, G. Wagner, N. Guarino, and M. v. Sinderen, "An Ontologically Well-Founded Profile for UML Conceptual Models," in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, A. Persson and J. Stirna, Eds. Springer Berlin Heidelberg, Jun. 2004, no. 3084, pp. 112–126. ISBN 978-3-540-22151-7, 978-3-540-25975-6. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-25975-6_10

[23] Z. Rybola and K. Richta, "Transformation of Special Multiplicity Constraints - Comparison of Possible Realizations," in *Proceedings of the Federated Conference on Computer Science and Information Systems*, FedCSIS 2012, Wroclaw, Poland, Sep. 2012. ISBN 978-83-60810-51-4 pp. 1357–1364.

[24] ——, "Possible Realizations of Multiplicity Constraints," *Computer Science and Information Systems*, vol. 10, no. 4, pp. 1621–1646, Oct. 2013. doi: 10.2298/CSIS121210067R

[25] R. Pergl, T. P. Sales, and Z. Rybola, "Towards OntoUML for Software Engineering: From Domain Ontology to Implementation Model," in *Model and Data Engineering*, ser. LNCS, vol. 8216. Amantea, Italy: Springer, Sep. 2013. doi: 10.1007/978-3-642-41366-7. ISBN 978-3-642-41365-0 pp. 249–263.

[26] W. Kuskorn and S. Lekcharoen, "An Adaptive Translation of Class Diagram to Relational Database," in *International Conference on Information and Multimedia Technology, 2009. ICIMT '09*, Dec. 2009. doi: 10.1109/ICIMT.2009.56 pp. 144–148.

[27] P. Rob and C. Coronel, *Database Systems: Design, Implementation, and Management*, 2nd ed. Boyd & Fraser, 1995. ISBN 0-7895-0052-3

[28] Z. Rybola and K. Richta, "Transformation of Binary Relationship with Particular Multiplicity," in *DATESO 2011*, vol. 11. Písek, Czech Republic: Department of Computer Science, FEECS VSB - Technical University of Ostrava, Apr. 2011. ISBN 978-80-248-2391-1 pp. 25–38.