Computer Science and Information Systems 14(3):913–937 DOI: https://doi.org/10.2298/CSIS170109035R

Towards OntoUML for Software Engineering: Transformation of Kinds and Subkinds into Relational Databases

Zdeněk Rybola and Robert Pergl

Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 16000 Praha 6
{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. While Model-Driven Development is a well-established approach, OntoUML has been overlooked so far as a conceptual modelling language for the PIM of application data. This paper is an extension of the paper presented at MDASD 2016, where we outlined the transformation of Rigid Sortal Types – Kinds and Subkinds. In this paper, we discuss the details of various variants of the transformation of these types and the rigid generalization sets. The result of our effort is a complete method for preserving high-level ontological constraints during the transformations, specifically special multiplicities and generalization set meta-properties in a relational database using views, CHECK constraints and triggers.

Keywords: OntoUML, UML, transformation, relational database, Kind, Subkind, generalization set.

1. Introduction

Software engineering is a demanding discipline that deals with complex systems [8]. 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. One of these approaches is Model-Driven Development (MDD), which is based on elaborating models and transformations between them [20]. The most usual part of the MDD approach used in the practice is the process of *forward engineering*: transformations of more abstract models (e.g. a PIM (platform independent model)) into more specific ones (e.g. PSM (platform specific model) or ISM (implementation specific model)). The most common use-case of such a 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 [8]. To successfully use them in the MDD approach, the models 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 [10]. OntoUML seems to be a very suitable language for modelling platform-independent data models. As it is based on Unified Foundational Ontology (UFO), it is domain-agnostic and it provides mechanisms to create ontologically well-founded conceptual models [10].

In this paper, we discuss introduction of OntoUML into MDD, specifically we formulate two research questions:

1. Are there benefits of using OntoUML Kinds and Subkinds in PIM?
2. Is it possible to preserve generalization sets constraints of Kinds and Subkinds in a relational database?

The first research question is motivated by success of OntoUML for making precise ontological models and thus exploring the possibility of incorporating OntoUML into the MDD as a primary language for conceptual data modelling. As relational databases are still the most popular type of data storage[1], we focus on modelling OntoUML PIMs and their transformation into their realization in relational databases. One of the key challenges in MDD is preserving model constraints during transformations (the second research question).

To elevate the current knowledge of model transformation and database modelling, the transformation of the OntoUML PIMs is divided into three consecutive steps:

1. transformation of the OntoUML PIM into UML PIM;
2. transformation of the UML PIM into RDB PSM (PSM of relational database);
3. transformation of the RDB PSM into SQL ISM (SQL scripts for database schema creation) [28].

This paper is part of a series discussing the transformation of various types of universals used in OntoUML into their correct and complete realization in the RDB ([28], [29], [30]). In particular, this paper is an extended version of the paper presented at MDASD 2016 [30], where the transformation of Rigid Sortal types – Kinds and Subkinds – was introduced. In this paper, we elaborate deeply on various possibilities of the realization during the second and third step of the transformation, focusing on the realization of meta-properties of the generalization sets. We illustrate the possibilities on the running example discussed in section 4.

The structure of the paper is as follows: in section 2, the work related to our approach is discussed; in section 3, the relevant constructs and principles of OntoUML and UFO for this contribution are introduced; in section 4, the running example used for the demonstration of our approach is introduced; in section 5, our approach is discussed and illustrated on the running example; in section 6, discussion to our approach is provided; finally, in section 7, the conclusion of the paper results is provided.

2. Background

2.1. Previous work

Our approach was introduced in [28], where we presented the idea of the three-step transformation using UML and OCL constraints as the intermediate steps. We also discussed

[1] According to ranking published on `https://db-engines.com/en/ranking`, 7 of 10 most popular database systems are relational.

the possibility to use the approach discussed in [32] for the realization of the OCL constraints derived during the transformation of the OntoUML universal types. In [29], we presented more details about the transformation of anti-rigid Sortal universal types (Roles and Phases), discussing the options for the realization of the anti-rigid relations including special OCL constraints and their realization in the RDB using database views.

Finally, in [30], we discussed in more details the transformation of rigid Sortal universal types – Kinds and Subkinds, describing the possible realizations, the derived OCL constraints realizing the meta-properties of the generalization sets in the RDB PSM and their realization in the RDB using database views, CHECK constraints triggers. This paper is an extended version of the paper [30], discussing details of possible realizations and their consequences and providing more examples (especially for the triggers missing in the initial paper).

2.2. Related work

We may distinguish several efforts dealing with the Impedance Mismatch Problem (IMP) [37] of conceptual models and the relational model. These are:

1. Transformation of traditional Entity-Relationship (ER) models into the relational model.
2. Using the UML notation to express relational models.
3. Object-relational mapping technologies (ORM).
4. Transformation from (Onto)UML into a relational model: UML PIM into an RDB PSM

Ad 1 is a long-studied and well-established approach documented e.g. in [27]. We use these approaches in our work. There are also approaches for transformation of the Extended ER models (EER), e.g. [5]. However, the traditional approaches neglect checking certain constraints. As discussed later, when realizing generalization sets and their meta-properties in RDB, the existence of referencing records in other tables must be checked. Similar problem is also addressed in [1] as *inverse referential integrity constraints* (IRICs), where the authors present an approach to the automated implementation of the IRICs by database triggers in a tool called IIS*Case. This tool was designed to provide a complete support for developing database schemas including the check of the consistency of constraints embedded into the DB [18] and the integration of subschemas into a relational DB schema [17]. The transformation of constraints specifically has been elaborated e.g. in [18] and [23]. Next, Rybola and Richta discuss implementation of special multiplicity constraints in [32], which we also use in our work.

There are several approaches to ad 2, but we do not refer to them further here, as they actually do not deal with a conceptual transformation and are mentioned just for the sake of completeness. Ad 3 are technologies offered by libraries of various object-oriented languages (such as Java, Smalltalk, Ruby and C#) to overcome the object-relational Impedance Mismatch Problem (IMP). The library routines perform an automatic run-time transformation between the object model and the relational model. An extensive study of the current leading ORM solutions is presented by Torres et al. in [37]. Nevertheless, ORM works at an application level, while our goal is to push richer semantics to the database level. Also, as noted by Torres et al., ORM provides tools to work with the IMP, but not a complete methodology to solve them.

Ad 4 are approaches most similar to our effort. In [16], the authors describe a tool called Dresden OCL Toolkit [6] which is able to validate a model instance against defined OCL constraints. The tool can be also used to generate SQL code from the model and attached OCL constraints, realizing the constraints using database views. We inspired by this approach in one of our proposed realizations of the generalization set constraints (as well as other types of constraints not discussed here) derived from the initial OntoUML PIM. Another related work can be found in [7], where the authors transform OCL constraints into stored procedures. However, we prefer using other techniques, as procedures must be invoked explicitly by the application, while the suggested CHECK constraints and triggers are executed automatically when performing standard DML operations. Also, we present the realization specially for Oracle Database 12g, while the authors of [7] discusses MariaDB, PostgreSQL and SQL server. Another approach to the realization of UML PIM is discussed e.g. in [22] or [38], where the authors present transformation of a UML PIM into an Object-Relational database. Also, they do not discuss realization of the meta-properties of generalization sets nor other types of constraints, which can be derived from an OntoUML PIM, which is the focus of our research.

As for the transformation of OntoUML specifically, based on the literature review and personally confirmed by the author of OntoUML Dr. Giancarlo Guizzardi, there is no published method for transformation into UML apart from our previous work so far. Instead, there are works dealing with transformation of OntoUML into different languages such as OWL [39] and Alloy [4].

2.3. UML

As OntoUML is based on UML and UML is used in the intermediate steps of the transformation, certain aspects of the language require to be outlined. UML (Unified Modeling Language) [25] is a popular modelling language for creating and maintaining variety of models using diagrams and additional components [34]. In context of the data modelling, UML Class Diagram is the notation mostly used to define conceptual models of application data [2]. Also, to describe the structure of a relational database schema, UML Data Modelling profile as an extension to the UML Class Diagrams may be used [35].

The main elements of a UML Class Diagram are classes, which serve to classify various types of objects of the domain and specify the features and behaviour of their instances. The classes can be related by various types of connectors to define relations between the classes or their instances. In context of this paper, the generalization/specialization relation is important. It is used between the classes to inherit features form a superclass (more abstract concept) to the subclasses (more specific concepts) [25]. As UML is designed following the object-oriented programming approach, an object can be an instance of only one class [2], although according to *Liskov substitution principle*, an instance of a subclass can be used on any place where instance of its superclass is expected [19].

The subclasses of the same superclass may form a *generalization set* to define a partition of subclasses with common meaning [25]. 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 individual versions of UML: UML 2.4.1and older define the {`incomplete, disjoint`} as default, while UML 2.5 defines the {`incomplete, overlapping`} as default. However, as each object is an instance of exactly one class in the most current programming languages, the concept of generalization sets can only be used in conceptual models and it must be transformed before its actual realization.

To define additional constraints in the UML models, Object Constraint Language (OCL) [24] is used. OCL is a specification language, which is part of the UML standard. In OCL, it is possible to define various conditions, which must be satisfied by all instances of contextual classes, and many other types of constraints. In our approach, we are using OCL to define constraints in the UML PIM and PSM of the relational database.

3. OntoUML

OntoUML is a conceptual modelling language focused on building ontologically well-founded models. It was formulated in Guizzardi's PhD Thesis [10] as a light-weight extension of UML based on UML profiles.

The language is based on *Unified Foundational Ontology* (UFO) [14], 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 [12] or roles and the counting problem [11]. The language has been successfully applied in different domains such as interoperability for medical protocols in electrophysiology [9] and the evaluation of an ITU-T standard for transport networks [3].

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.

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`) [10].

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) [14].

3.2. Identity and Identity Principle

Identity is one of the key features of UFO. Identity is the fact of being what an individual is, enabling distinction of different individuals. Identity principle, on the other hand, defines the methods to determine, if two apparent instances of the same concept are actually the very same individual. 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* [14] or *Dispersive universals* [13] (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.

3.3. 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 [15]. UFO distinguishes *rigid* (instances of rigid universals are their instances in all worlds), *anti-rigid* (each instance of an anti-rigid universal from any world is not its instance in certain other world(s)) and *semi-rigid* (some instances of semi-rigid universals are always their instances, other instances may not be their instances in certain worlds) universals [10].

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

3.4. 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 subtype is also an instance of the supertype automatically by following the same identity principle, and thus it also receives the properties defined by the supertype. Additionally, the relation is rigid in UML – an instance of the subclass can always be used as an instance of the superclass unless it ceases to exist – 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 [10].

Although not very common in UML models, the generalization sets are crucial in OntoUML models as they define the required identity for various universal types. Unless altered, $\{$*incomplete, non-disjoint*$\}$ is considered the default value of the meta-properties, which is in contrast to UML 2.4.1 and earlier.

3.5. 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 [14]. In OntoUML, the Kind universals are depicted as classes with the $\ll Kind \gg$ stereotype [15].

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 [14]. In OntoUML, the Kind universals are depicted as classes with the $\ll Subkind \gg$ stereotype [15].

4. Running Example

Our approach to the transformation of Kinds, Subkinds and their generalization sets from the OntoUML PIM into SQL ISM is illustrated on the running example shown in Figure 1. The model shows an excerpt of the domain of transportation company. The company uses various vehicles for transportation of various types of load, ranging from persons to documents to heavy cargo. Therefore, the main entity of such model is the type `Vehicle`. As it is the type defining the identity principle for its instances, it is classified as *Kind*. For each vehicle used by the company, its manufacturer, model and plate number is needed, represented by the respective attributes of the `Vehicle` type.

Furthermore, the company distinguishes three special types of vehicles – trucks, cars and motorcycles – for which additional attributes are important. Each of these types of vehicles is represented by its own type in the OntoUML PIM with the appropriate attributes. As all these types represent the specialization of the general concept of a vehicle, their are classified as *Subkinds* forming a generalization set specializing the type `Vehicle`. Moreover, as the types of the vehicles are disjoint – clearly, one vehicle cannot be a truck and a motorcycle at the same time – the generalization set is defined `disjoint`. On the other hand, there might be other types of vehicles, for which no special attributes need to be recorded. Therefore, the generalization set is not defined `complete`, but rather left `incomplete`.

5. 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:

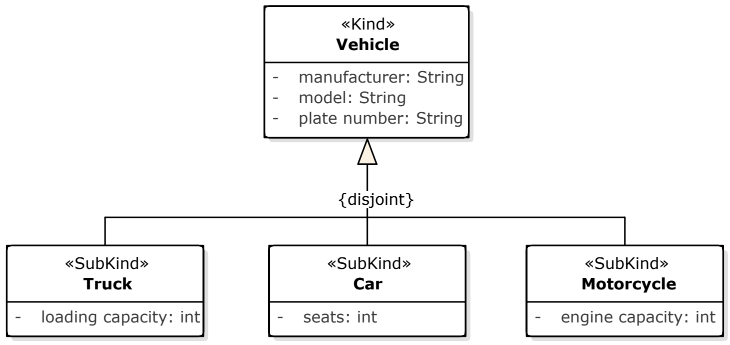

Fig. 1. Example of an OntoUML model with Kinds and Subkinds

1. subsection 5.1 discusses the transformation of an OntoUML PIM into a UML PIM,
2. subsection 5.2 discusses the transformation of the UML PIM into an RDB PSM,
3. subsection 5.3 discusses the transformation of the RDB PSM into an SQL ISM.

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 constraints for meta-properties of generalization sets, these constraints should be carried over to the other models. However, as generalization sets are not very common in UML models (although defined by the standard), these constraints are not addressed in the common transformation approaches. Therefore we focus on correct and complete realization of these generalization sets including their meta-properties. Whenever not possible to express a constraint directly in the diagrams, we use OCL to define such constraint.

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. [27,32] and tools for transformation of a UML PIM into database models such as Enterprise Architect), while the direct transformation would have to be built from the scratch. Also, various optimizations and refactoring may be applied whenever possible (e.g. when the subclasses hold no attributes and are used only to distinguish various subtypes of the superclass entity, they can be expressed by values of a single attribute of the superclass).

To demonstrate our approach, we refer to the technical report [31], where all the OCL constraints and SQL scripts are defined for the running example discussed in section 4. We use Oracle Database 12c SQL dialect [26] in all the examples in the SQL ISM.

In the approach presented here, we assume the (most common) situation where all attributes of the model classes have multiplicities `[1..1]`. In the discussion in section 6, we discuss how the situation changes for different multiplicities.

5.1. Transformation of OntoUML PIM into UML PIM

In the first step of the transformation, the OntoUML model is transformed into pure UML model, transforming the types in the OntoUML PIM into the classes in the UML PIM while preserving all the semantics defined by the particular universals of the types.

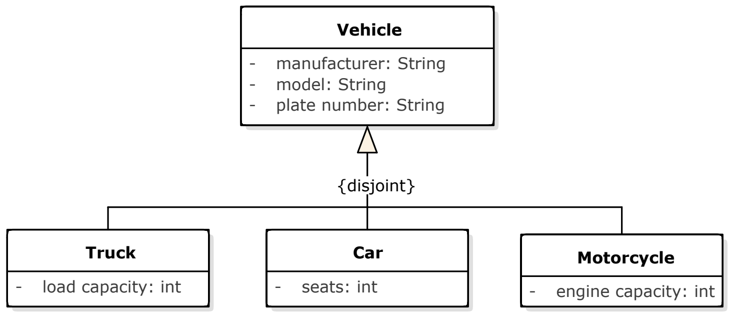

Fig. 2. UML PIM with the transformed Kinds and Subkinds

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

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 classes and their instances. Therefore, the representation of Kinds and Subkinds in UML may stay the same: each $\ll Kind \gg$ and $\ll Subkind \gg$ class from the OntoUML PIM is transformed into a standard UML class in the UML PIM keeping all its features – attributes and relations.

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

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 – because of the inherited identity principle and it cannot cease to be the instance of any of them without loosing its identity.

Thanks to this rigidity, the generalization sets of $\ll Subkind \gg$ classes in the OntoUML PIM can be transformed into standard UML generalization sets in the UML PIM. Also, the meta-properties `isDisjoint` and `isCovering` of the generalization set remain the same – the only difference may be the need to show these properties, depending on the version of UML as discussed in section 2. The example of this transformation can be seen in Figure 2.

5.2. Transformation of PIM into PSM

The second step is the transformation of the UML PIM into RDB PSM. The UML Data Model profile – an extension to the UML class diagrams – is used in the examples to define the structure of the relational database in UML [35]. Additional constraints required to preserve the semantics derived during the transformation of 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 [6].

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 references restricted by FOREIGN KEY constraints. Also, PRIMARY KEY constraints are defined for unique identification of individual records in the tables [32]. Therefore, the transformation of classes representing various Kind universals is straightforward – the class with its attributes is transformed into a table with its columns.

However, the concept of generalization is not present in relational databases. Therefore, the generalization sets must be transformed into into tables and references. In general, there are three approached commonly used for the realization of the generalization in relational database [27]:

- by a single table containing columns for all the attributes of the superclass and all the subclasses;
- by individual tables for each of the subclasses, containing the columns for the attributes of the respective subclass and the superclass;
- by a table for the superclass and individual tables for all the subclasses referencing the superclass table.

Each of these variants brings certain limitations and consequences. However, in any case, the generalization set constraints *isCovering* and *isDisjoint* defined in the UML PIM, as well as the multiplicity constraints of the attributes, should be realized as well to preserve the constraints explicitly defined in the model. However, these properties are usually not truly realized in the database. Therefore, we focus on the possibilities for their realization in the RDB PSM to truly preserve the semantics defined in the original OntoUML PIM.

In the following subsections, the details of each of the possible realizations are discussed in context of the generalization set constraints and their realization, as well as other limitations and consequences.

Single Table. In this variant, the superclass and all its subclasses are realized by a single table, defining the columns for all the attributes of the superclass and all its subclasses. Instances of the superclass are represented by rows with NULL values in the columns of the subclasses. Instances of a subclass contain values only in the superclass columns and the columns of that subclass – the columns of the other subclasses contain NULL values. Additionally, adding an extra *discriminator* column to discriminate the subclasses may be convenient [27]. An example of this realization of the generalization set in the running example is shown in Figure 3.

In this realization, the mandatory values of the supeclass's attributes can be easily realized by NOT NULL constraints, as all instances always have values for such mandatory attributes. However, the mandatory values of the subclasses's attributes cannot be realized by such simple constraints, as they are dependent on the actual type of the stored instance.

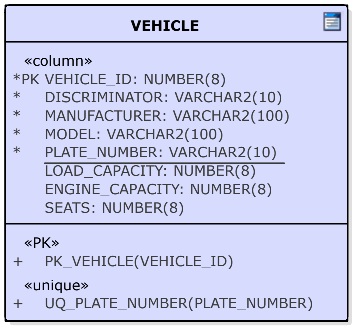
VEHICLE
«column»
*PK VEHICLE_ID: NUMBER(8)
* DISCRIMINATOR: VARCHAR2(10)
* MANUFACTURER: VARCHAR2(100)
* MODEL: VARCHAR2(100)
* PLATE_NUMBER: VARCHAR2(10)
LOAD_CAPACITY: NUMBER(8)
ENGINE_CAPACITY: NUMBER(8)
SEATS: NUMBER(8)
«PK»
+ PK_VEHICLE(VEHICLE_ID)
«unique»
+ UQ_PLATE_NUMBER(PLATE_NUMBER)

Fig. 3. RDB PSM with the generalization set realized by a single table

Instead, these constraints must be defined with the relation to the value in the *discriminator* column, which in turn can contain only values representing the appropriate classes of the original generalization set according to its meta-properties. Therefore, the following constraints should be realized:

– The *discriminator* value must match one of the subclasses. In the case of an `incomplete` generalization set, also the superclass is allowed. In the case of an `overlapping` generalization set, also all the combinations of the subclasses are allowed.
– All mandatory columns of the subclass(es) identified by the *discriminator* values must contain NOT NULL values, while the columns of all the other subclasses must contain NULL values. In the case of identifying the superclass, all columns of all the subclasses must contain NULL values.

As discussed above, the NOT NULL constraints for the columns representing the attributes of the superclass can be defined directly on the column level, as all instances will always provide values for these columns. However, the constraints for the columns representing the attributes of the subclasses follow complex rules, which cannot be defined on the column level. Instead, they must be defined on the table level by additional constraints. In our approach, we use OCL invariants to define such constraints in the RDB PSM. An example of the OCL constraint realizing the {`disjoint,incomplete`} generalization set from the running example realized by a single table is shown in Constraint 1. Examples of the realization of the other variants of the generalization set meta-properties can be found in [31].

In case that values of some of the attributes of the superclass or some of the subclasses should be unique, the realization of such constraint in the RDB PSM is simple. As all data are stored in the same table, the column representing the particular attribute can be simply restricted by the UNIQUE constraint, as shown in Figure 3 where the uniqueness is defined by the `UQ_PLATE_NUMBER` constraint.

Individual Tables. In this variant, a table is created for each possible type of instance from the generalization set. Usually, it means creating a table for each of the subclasses,

Constraint 1 OCL invariant for the {`disjoint,incomplete`} generalization set realized by a single table

```
context v:VEHICLE inv GS_Vehicle_Types:
def Vehicle_Instance: Boolean =
  v.DISCRIMINATOR = 'Vehicle' AND v.LOAD_CAPACITY = OclVoid
    AND v.SEATS = OclVoid AND v.ENGINE_CAPACITY = OclVoid
def Truck_Instance: Boolean =
  v.DISCRIMINATOR = 'Truck' AND v.LOAD_CAPACITY <> OclVoid
    AND v.SEATS = OclVoid AND v.ENGINE_CAPACITY = OclVoid
def Car_Instance: Boolean =
  v.DISCRIMINATOR = 'Vehicle' AND v.LOAD_CAPACITY = OclVoid
    AND v.SEATS <> OclVoid AND v.ENGINE_CAPACITY = OclVoid
def Motorcycle_Instance: Boolean =
  v.DISCRIMINATOR = 'Motorcycle' AND v.LOAD_CAPACITY = OclVoid
    AND v.SEATS = OclVoid AND v.ENGINE_CAPACITY <> OclVoid

Vehicle_Instance OR Truck_Instance OR Car_Instance OR Motorcycle_Instance
```

containing columns for the attributes of the superclass and the particular subclass [27]. However, to correctly realize the meta-properties of the generalization set, a table for the superclass should be also created in the case of an `incomplete` generalization set and a table for each of the combinations of the subclasses in the case of an `overlapping` generalization set. An example of such realization of the running example is shown in Figure 4.

With this realization, in each of the tables, the mandatory attributes can be easily realized by NOT NULL constraints defined for all the columns representing the mandatory attributes of both the superclass and the particular subclass(es), which the table represents, as only instances of the same combination of classes are stored in a single table, and thus they always have values for those attributes.

However, on the other hand, it is more complicated to ensure the unique values for attributes of the superclass, when needed. The same also applies for attributes of subclasses in the case of an `overlapping` generalization set. It is because the values of such attributes are distributed among multiple tables representing the particular combination of classes from the generalization set. For instance, in the running example, the values of the `plate number` attribute of all vehicles should be unique, regardless of its actual type. As it is realized by the `PLATE_NUMBER` column in all the tables shown in Figure 4, it is not possible to simply restrict the columns only by a simple UNIQUE constraint. Such constraint – named `UQ_PLATE_NUMBER` in each of the tables – only ensures the uniqueness of the values in that particular table. Instead, an additional constraint must be defined to ensure the uniqueness of the values across all the tables with the following properties:

- A constraint is generated for each of the tables realizing the generalization set.
- Each constraint checks, that there is no record in all the other tables with the same value of the unique column as in the constrained table.

As such constraint cannot be defined directly in the table, we use OCL to define them. An example of such OCL constraint for the table `VEHICLE` realiting the unique constraint of the `PLATE_NUMBER` column distributed across the tables shown in Figure 4 is shown in Constraint 2. The constraints for the other tables can be found in [31].

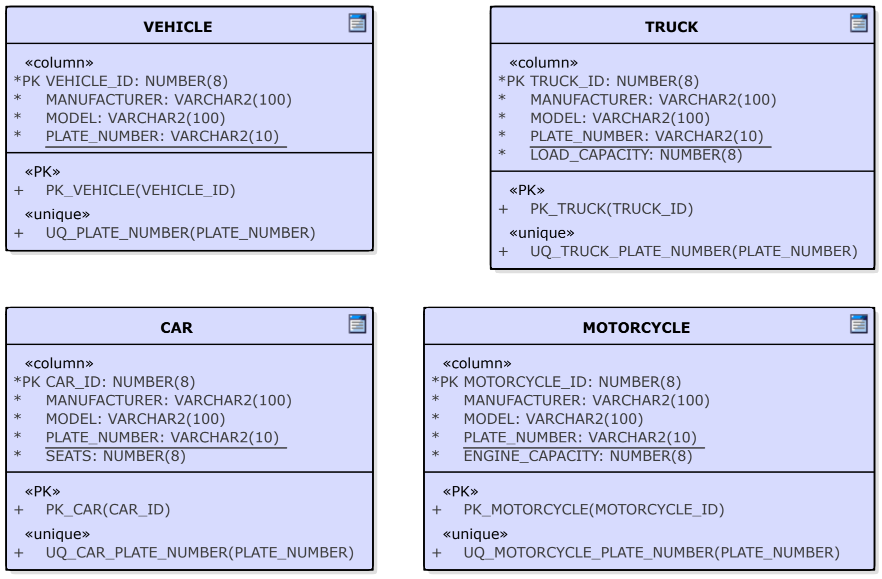
VEHICLE
«column»
*PK VEHICLE_ID: NUMBER(8)
* MANUFACTURER: VARCHAR2(100)
* MODEL: VARCHAR2(100)
* PLATE_NUMBER: VARCHAR2(10)
«PK»
+ PK_VEHICLE(VEHICLE_ID)
«unique»
+ UQ_PLATE_NUMBER(PLATE_NUMBER)
TRUCK
«column»
*PK TRUCK_ID: NUMBER(8)
* MANUFACTURER: VARCHAR2(100)
* MODEL: VARCHAR2(100)
* PLATE_NUMBER: VARCHAR2(10)
* LOAD_CAPACITY: NUMBER(8)
«PK»
+ PK_TRUCK(TRUCK_ID)
«unique»
+ UQ_TRUCK_PLATE_NUMBER(PLATE_NUMBER)
CAR
«column»
*PK CAR_ID: NUMBER(8)
* MANUFACTURER: VARCHAR2(100)
* MODEL: VARCHAR2(100)
* PLATE_NUMBER: VARCHAR2(10)
* SEATS: NUMBER(8)
«PK»
+ PK_CAR(CAR_ID)
«unique»
+ UQ_CAR_PLATE_NUMBER(PLATE_NUMBER)
MOTORCYCLE
«column»
*PK MOTORCYCLE_ID: NUMBER(8)
* MANUFACTURER: VARCHAR2(100)
* MODEL: VARCHAR2(100)
* PLATE_NUMBER: VARCHAR2(10)
* ENGINE_CAPACITY: NUMBER(8)
«PK»
+ PK_MOTORCYCLE(MOTORCYCLE_ID)
«unique»
+ UQ_MOTORCYCLE_PLATE_NUMBER(PLATE_NUMBER)

Fig. 4. RDB PSM with the generalization set realized by individual tables

Constraint 2 OCL invariants for distributed unique column `PLATE_NUMBER`

```
context v:VEHICLE inv UQ_Vehicle_Plate_number:
NOT(TRUCK.allInstances()->exists (t|t.PLATE_NUMBER = t.PLATE_NUMBER))
 AND
  NOT(CAR.allInstances()->exists (c|c.PLATE_NUMBER = t.PLATE_NUMBER))
 AND
  NOT(MOTORCYCLE.allInstances()->exists (m|m.PLATE_NUMBER = t.PLATE_NUMBER))
```

Related Tables. In this variant, all the classes are transformed into their own respective tables. Each table contains only attributes of the particular class and the PRIMARY KEY. Additionally, the subclass tables also contain reference to the superclass table restricted by the FOREIGN KEY constraint. Also, a special *discriminator* column in the superclass table to discriminate the actual type of the instance is convenient. The instances of the classes from the UML PIM are then stored in the appropriate combination of tables. Moreover, as the references realize generalization, the references are always one-to-one. Therefore, the reference should be unique and can be combined with the PRIMARY KEY column [27]. An example of this realization of the generalization set in the running example is shown in Figure 5.

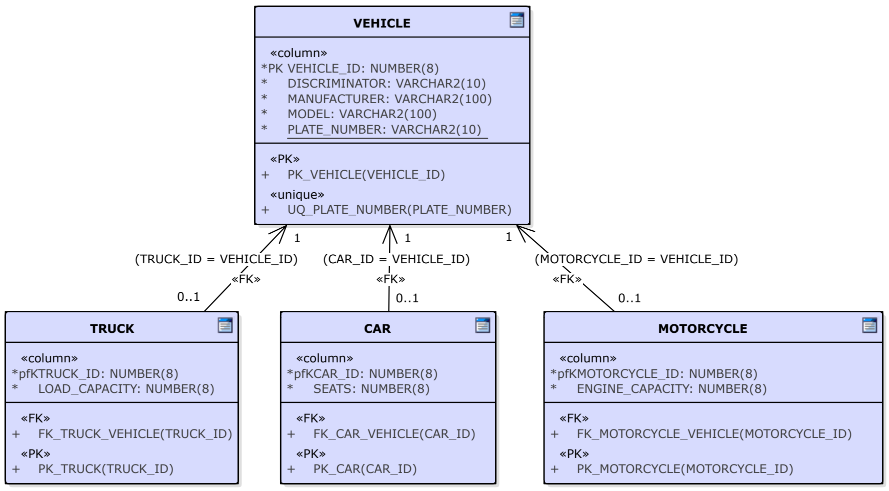
VEHICLE
«column»
*PK VEHICLE_ID: NUMBER(8)
* DISCRIMINATOR: VARCHAR2(10)
* MANUFACTURER: VARCHAR2(100)
* MODEL: VARCHAR2(100)
* PLATE_NUMBER: VARCHAR2(10)
«PK»
+ PK_VEHICLE(VEHICLE_ID)
«unique»
+ UQ_PLATE_NUMBER(PLATE_NUMBER)
1
1
1
(TRUCK_ID = VEHICLE_ID)
«FK»
0..1
(CAR_ID = VEHICLE_ID)
«FK»
0..1
(MOTORCYCLE_ID = VEHICLE_ID)
«FK»
0..1
TRUCK
«column»
*pfKTRUCK_ID: NUMBER(8)
* LOAD_CAPACITY: NUMBER(8)
«FK»
+ FK_TRUCK_VEHICLE(TRUCK_ID)
«PK»
+ PK_TRUCK(TRUCK_ID)
CAR
«column»
*pfKCAR_ID: NUMBER(8)
* SEATS: NUMBER(8)
«FK»
+ FK_CAR_VEHICLE(CAR_ID)
«PK»
+ PK_CAR(CAR_ID)
MOTORCYCLE
«column»
*pfKMOTORCYCLE_ID: NUMBER(8)
* ENGINE_CAPACITY: NUMBER(8)
«FK»
+ FK_MOTORCYCLE_VEHICLE(MOTORCYCLE_ID)
«PK»
+ PK_MOTORCYCLE(MOTORCYCLE_ID)

Fig. 5. RDB PSM with the generalization set realized by related tables

The mandatory attributes of the superclass and the subclasses can be simply realized by NOT NULL constraints, as each table always contains the same portion of data of instances of the same type. Also, the UNIQUE constraints for the individual attributes of the superclass and subclasses can be easily realized by UNIQUE constraints, as each attribute is realized only by a single column in a single table – see the `UQ_PLATE_NUMBER` constraint defined in the `VEHICLE` table.

To correctly realize the meta-properties of the generalization set, the existence of referencing records in the subclass tables should be checked in context of the value in the *discriminator* column of the superclass table, which in turn can contain only values representing the appropriate classes or their combination from the original generalization set according to its meta-properties. Therefore, the following constraints should be realized:

– The *discriminator* value in the superclass table must match one of the subclasses. In the case of an `incomplete` generalization set, also the superclass is allowed. In the case of an `overlapping` generalization set, also all the combinations of the subclasses are allowed.

Constraint 3 OCL invariant for the {`disjoint,incomplete`} generalization set realized by related tables

```
context v:VEHICLE inv GS_Vehicle_Types:
def Vehicle_Instance: Boolean = v.DISCRIMINATOR = 'Vehicle'
    AND NOT (TRUCK.allInstances()->exists (t|t.TRUCK_ID = v.VEHICLE_ID))
    AND NOT (CAR.allInstances()->exists (c|c.CAR_ID = v.VEHICLE_ID))
    AND NOT (MOTORCYCLE.allInstances()->exists (m|m.MOTORCYCLE_ID = v.VEHICLE_ID))
def Truck_Instance: Boolean = v.DISCRIMINATOR = 'Truck'
    AND TRUCK.allInstances()->exists (t|t.TRUCK_ID = v.VEHICLE_ID)
    AND NOT (CAR.allInstances()->exists (c|c.CAR_ID = v.VEHICLE_ID))
    AND NOT (MOTORCYCLE.allInstances()->exists (m|m.MOTORCYCLE_ID = v.VEHICLE_ID))
def Car_Instance: Boolean = v.DISCRIMINATOR = 'Car'
    AND ...
def Motorcycle_Instance: Boolean = v.DISCRIMINATOR = 'Motorcycle'
    AND ...

Vehicle_Instance OR Truck_Instance OR Car_Instance OR Motorcycle_Instance
```

– For each record in the superclass table, there is a single referencing record in each of the tables identified by the *discriminator* value and no referencing record in all the other tables. In the case of identifying the superclass, there are no referencing records in all the subclass tables.

As such constraints restrict data in multiple tables, they cannot be defined on the column level like other database constraints. Instead, a special constraint on the table level must be defined. We use OCL in our approach. Furthermore, as the references are based on the value of the *discriminator* column, covering all its possible values, both constraints can be realized by a single OCL invariant. An example of such constraint for the {`disjoint,incomplete`} generalization set from the running example in Constraint 3. Examples of the realization of the other variants of the generalization set meta-properties can be found in [31].

5.3. Transformation of PSM into ISM

The last step is the transformation of the RDB PSM into SQL ISM. This model consists of database scripts for the creation of the database tables, their 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 [36] can be used to generate SQL DDL scripts. These scripts usually include the CREATE statements 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 [6].

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 [32]. 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 when inserting, updating or removing data, however, they do not actually prevent inserting data violating the constraints. It is still possible to get invalid data into the database and query them using standard SELECT statements over the original tables. It is necessary to ensure the correct usage of the views on the application level.
– *Updatable database views with CHECK option* can be used to query only the valid data. Moreover, they can be used to manipulate the data by DML operations like INSERT, UPDATE and DELETE, as well. The CHECK option prevents creating a record which will not be accessible by the view, however, it does not prevent changing data in other tables which might violate the constraint. Moreover, the updatable views are restricted by several constraints for the query expression in their definition, as discussed in [32].
– *CHECK constraints* can be used to check the values inserted to various columns of the table. Any DML operation affecting the value in such a column is then validated against the CHECK constraint and rolled back, if the constraint is violated. Unfortunately, the common contemporary database engines (e.g. Oracle 11g) do not support subqueries in the CHECK constraint statements. Therefore, they can be effectively used to realize constraints restricting data in a single table, but not for the relational constraints restricting the records in related tables.
– *Triggers* can be defined to perform complex data validations and manipulations when various DML operations are executed on a table. Being able to define complex queries in the trigger body, they are capable to deal with almost every possible constraint and prevent any operation which would violate them. The constraint checks in the triggers slow down each constrained DML operation, however as shown in [32], the time increase is typically not substantial. On the other hand, it is possible to entirely prevent creating invalid data in the database and save a lot of checking implementation on the application level.

In [32], 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 as discussed in the following sections. As the generalization set in the running example is {`disjoint,incomplete`}, only realization of such OCL constraints shown in subsection 5.2 is discussed. However, the other variants would be realized similarly.

Single table. When the generalization set is transformed into a single database table, a special OCL constraint is defined to ensure the meta-properties of the generalization set by checking the values in the columns of the particular subclasses according to the *discriminator* value. An example of the constraint is shown in Constraint 1.

As the constraint simply restricts values of columns in a single table to be empty or non-empty, it can be easily realized in the SQL ISM by a *database view* with a query selecting only records with NULL and NOT NULL values in the appropriate columns according to the *discriminator* value. Such *database view* can be used to query only valid records, ignoring all records violating the constraint. Moreover, as the view meets the condition to be defined as *updatable view* WITH CHECK OPTION, it can also be used for the DML operations, preventing creating invalid data in the original table. Still, the

SQL 1 Updatable database view to query valid data from the combined `Vehicle` table

```
CREATE VIEW GS_VEHICLE_TYPES_VIEW AS
SELECT * FROM VEHICLE v WHERE
  (v.DISCRIMINATOR = 'Vehicle' AND v.LOAD_CAPACITY IS NULL
    AND v.SEATS IS NULL AND v.CONTENT IS NULL)
  OR
  (v.DISCRIMINATOR = 'Truck' AND v.LOAD_CAPACITY IS NOT NULL
    AND v.SEATS IS NULL AND v.CONTENT IS NULL)
  OR
  (v.DISCRIMINATOR = 'Car' AND v.LOAD_CAPACITY IS NULL
    AND v.SEATS IS NOT NULL AND v.CONTENT IS NULL)
  OR
  (v.DISCRIMINATOR = 'Motorcycle' AND v.LOAD_CAPACITY IS NULL
    AND v.SEATS IS NULL AND v.CONTENT IS NOT NULL)
WITH CHECK OPTION;
```

SQL 2 CHECK constraint for the combined `Vehicle` table

```
ALTER TABLE VEHICLE ADD CONSTRAINT GS_VEHICLE_TYPES_CHECK CHECK (
  (DISCRIMINATOR = 'Vehicle' AND LOAD_CAPACITY IS NULL
    AND SEATS IS NULL AND CONTENT IS NULL)
  OR
  (DISCRIMINATOR = 'Truck' AND LOAD_CAPACITY IS NOT NULL
    AND SEATS IS NULL AND CONTENT IS NULL)
  OR
  (DISCRIMINATOR = 'Car' AND LOAD_CAPACITY IS NULL
    AND SEATS IS NOT NULL AND CONTENT IS NULL)
  OR
  (DISCRIMINATOR = 'Motorcycle' AND LOAD_CAPACITY IS NULL
    AND SEATS IS NULL AND CONTENT IS NOT NULL));
```

original table can be directly accessed by the DML and query operations, and therefore such view cannot guarantee the consistency. An example of such database view for the running example is shown in SQL 1.

Similarly, the constraint can also be realized by a *CHECK constraint*, which is checked after each operation on the table. Thanks to that, it is able to completely ensure the data consistency according to the constraint. As the constraint condition checks only data of a single record in a single table, there is no problem with its implementation in the common relational database engines. An example of such CHECK constraint for the running example is shown in SQL 2.

Finally, the constraint might also be realized by a trigger. As the constraint only restricts data in a single table, only a single trigger would be needed. This trigger would be defined to be executed BEFORE INSERT and UPDATE operations, as only these operations can create invalid records. Moreover, as the constraint restricts values of only isolated records, the trigger can be executed for each row, checking only the affected row instead of all the data in the table. Such trigger would check the new values of the affected row in the individual columns and roll back the operation, if the values do not match the condition of the constraint. However, as the constraint can be realized more conveniently by the CHECK constraint or a checked updatable database view, using the trigger is not recommended. Still, an example of such trigger can be found in [31].

SQL 3 Database views to query valid data from the individual tables

```
CREATE VIEW UQ_VEHICLE_PLATE_NUMBER_VIEW AS
SELECT * FROM VEHICLE v WHERE (
  NOT EXISTS (SELECT 1 FROM TRUCK t WHERE t.PLATE_NUMBER = v.PLATE_NUMBER)
  AND NOT EXISTS (SELECT 1 FROM CAR c WHERE c.PLATE_NUMBER = v.PLATE_NUMBER)
  AND NOT EXISTS (SELECT 1 FROM MOTORCYCLE m WHERE m.PLATE_NUMBER = v.PLATE_NUMBER)
) WITH CHECK OPTION;
```

Individual tables. When the generalization set is transformed using the approach of individual tables, no special constraint is needed to realize the meta-properties of the generalization set. However, when a uniqueness of a superclass attribute (or even a subclass attribute in case of an `overlapping` generalization set) must be ensured, special distributed uniqueness OCL constraints such as shown in Constraint 2 must be realized.

Each of such constraints can be transformed into a *database view* querying records from the particular table meeting the condition of the OCL constraint – having such value of the restricted unique column, which does not exist in any of the other tables. This view can be used to query only valid record, ignoring all records in that particular table containing a non-unique value. An example of the view realizing the constraint shown in Constraint 2 is shown in SQL 3. Moreover, as the view meets the conditions for an updatable view, it is defined with the WITH CHECK OPTION and it can be also used for the DML operations while preventing violation of the constraint. As similar views are also defined for the other tables containing the constrained attribute, their combination can completely prevent creation of invalid data in the tables. Still, the original tables can be accessed directly, and therefore such views cannot guarantee the entire database consistency.

Similarly, the constraint could be also realized by CHECK constraints, checking the value of the affected record does not exist in the other tables. However, such CHECK constraint would require subqueries for checking the data in the other tables, and although valid according to the SQL:1999 specification, the contemporary database engines do not support such CHECK constraints. Therefore, this realization is not possible.

Finally, a trigger can also be used to entirely prevent creating invalid data in the database. As the constraint restricts using a value already used in another table, only the INSERT and UPDATE operations executed on each of the tables can create data violating the constraints. Therefore, a trigger is defined FOR EACH ROW on each of the tables after INSERT and UPDATE operations, trying to find the new value defined for the constrained column in the other tables and rolling back the operation, if a record in the other tables is found. An example of the trigger for the table `VEHICLE` in the running example is shown in SQL 3. The other triggers for the other tables can be found in [31].

Related tables. When the generalization set is transformed using the approach of related tables, the constraint is defined in context of the superclass table checking the existence of referencing records in the appropriate subclass tables, such as shown in Constraint 3. However, the realization of this constraint in the SQL ISM is mutually exclusive with the FOREIGN KEY constraint – the generalization set constraint requires referencing records in the subclass tables and the FOREIGN KEY constraint requires existence of the referenced record, but the records must be inserted one-by-one. Therefore, the FOREIGN

SQL 4 Triggers for the individual tables

```
CREATE OR REPLACE TRIGGER UQ_VEHICLE_PLATE_NUMBER_TRIGGER
AFTER INSERT OR UPDATE ON VEHICLE
FOR EACH ROW
DECLARE
  l_count NUMBER := 0;
BEGIN
  SELECT count(1) INTO l_count FROM DUAL WHERE (
     EXISTS (SELECT 1 FROM TRUCK t WHERE t.PLATE_NUMBER = :new.PLATE_NUMBER)
     OR EXISTS (SELECT 1 FROM CAR c WHERE c.PLATE_NUMBER = :new.PLATE_NUMBER)
     OR EXISTS (SELECT 1 FROM MOTORCYCLE m WHERE m.PLATE_NUMBER = :new.PLATE_NUMBER));

  IF l_count > 0 THEN raise_application_error
    (-20101, 'OCL constraint UQ_Vehicle_Plate_number violated!');
  END IF;
END;
```

SQL 5 Database view to query only valid data from the superclass of the related tables

```
CREATE OR REPLACE VIEW GS_VEHICLE_TYPES_VIEW AS
SELECT * FROM VEHICLE v WHERE
  (v.DISCRIMINATOR = 'Vehicle'
     AND NOT EXISTS (SELECT 1 FROM TRUCK t WHERE t.TRUCK_ID = v.VEHICLE_ID)
     AND NOT EXISTS (SELECT 1 FROM CAR c WHERE c.CAR_ID = v.VEHICLE_ID)
     AND NOT EXISTS (SELECT 1 FROM MOTORCYCLE m WHERE m.MOTORCYCLE_ID = v.VEHICLE_ID))
  OR (v.DISCRIMINATOR = 'Truck'
     AND EXISTS (SELECT 1 FROM TRUCK t WHERE t.TRUCK_ID = v.VEHICLE_ID)
     AND NOT EXISTS (SELECT 1 FROM CAR c WHERE c.CAR_ID = v.VEHICLE_ID)
     AND NOT EXISTS (SELECT 1 FROM MOTORCYCLE m WHERE m.MOTORCYCLE_ID = v.VEHICLE_ID))
  OR (v.DISCRIMINATOR = 'Car' AND ...)
  OR (v.DISCRIMINATOR = 'Motorcycle' AND ...))
WITH CHECK OPTION;
```

KEY should be defined DEFERRABLE to be checked at the end of the transaction instead of at the time of each operation [21].

The transformation of the OCL constraint into a *database view* is simple. The view queries only data from the table representing the superclass meeting the condition of the constraint – records which have referencing records in the appropriate subclass tables according to the *discriminator* value. This view can be used to query valid instances of the superclass (in case of `incomplete` generalization set) and it is also used in the JOIN queries to query complete data of valid instances of the individual subclasses (or their combination in case of `overlapping` generalization sets). An example of such view for the running example and the constraint shown in Constraint 3 is shown in SQL 5. Moreover, as the view meets the criteria for being updatable, it is defined with the WITH CHECK OPTION clause and can be used for the DML operations instead of the original superclass table, preventing creation of invalid record by such operations. However, the constraint can be also violated by DML operations on the subclass tables. Although some of these operations can be checked by additional updatable views, it is not possible to prevent all possible violating operations (e.g. inserting a referencing record into inappropriate subclass table), and also the original tables can still be accessed directly. Therefore, the realization by the views cannot guarantee the data consistency entirely.

Similarly, a CHECK constraints could be also defined with the same conditions, automatically preventing creation of invalid data. However, such CHECK constraints would require subqueries to check data in other tables, which, although valid according to the SQL:1999 specification, is not supported by the contemporary database engines. Therefore, this realization is not applicable.

The constraint can also be realized by a set of triggers checking the individual DML operations on the individual table able to cause violation of the constraint. The triggers verify the appropriate condition and rolls the operation back in the case of violation. In total, the following triggers are needed for the realization of such OCL constraint:

– **INSERT OR UPDATE ON the superclass table**: When inserting the data into the superclass table or updating them, referencing records in the appropriate subclass tables must exist according to the new *discriminator* value.
– **INSERT ON each subclass table**: When inserting a record into a subclass table, it should reference a non-existent record in the superclass table (thanks to the deferred FOREIGN KEY constraint, it should be inserted later while checking existence of appropriate referencing records). Otherwise, it necessarily must be an inappropriate or duplicate record.
– **UPDATE ON each subclass table**: When updating the data in a subclass table and changing its reference value, there must be no record in the superclass table referenced by the old reference value (such record would lose the required referencing record) nor the new reference value (such record should be inserted later to satisfy the insertion constraint discussed above).
– **DELETE ON each subclass table**: When deleting the data from a subclass tables, the referenced record in the superclass should not exist, otherwise it would lose the required referencing record.

All of these triggers can be defined BEFORE the particular operations, as it is possible to detect the constraint violation before really changing the data. Also, defining the triggers FOR EACH ROW is more efficient, as only the affected row can violate the constraint.

As presented above, a lot of triggers must be defined to realize the constraint. Also, the DML operations are slowed down by their execution as they query data from other tables for each of the affected rows. However, in contrast to the other realizations, they entirely prevent violation of the constraint, and thus ensuring the data consistency. An example of the trigger for the table `VEHICLE` from the running example is shown in SQL 6. The other triggers can be found in [31].

6. Discussion

This paper is a part of research on OntoUML-based MDE, however, specifically the topic of generalization sets is applicable in pure UML, as well, as the UML standard also incorporates them.

6.1. Efficiency

As mentioned above, our approach to the realization of the constraints derived from the meta-properties of rigid generalization sets is based on the approach discussed in [32]. In

SQL 6 Trigger for the INSERT and UPDATE operation on the superclass of the related tables realization

```
CREATE OR REPLACE TRIGGER GS_VEHICLE_TYPES_TRIGGER_VEHICLE
BEFORE INSERT OR UPDATE ON VEHICLE
FOR EACH ROW
DECLARE
  l_count NUMBER(1);
BEGIN
  SELECT COUNT(*) INTO l_count FROM DUAL WHERE (
    (:new.DISCRIMINATOR = 'Vehicle'
      AND NOT EXISTS (SELECT 1 FROM TRUCK t WHERE t.TRUCK_ID = :new.VEHICLE_ID)
      AND NOT EXISTS (SELECT 1 FROM CAR c WHERE c.CAR_ID = :new.VEHICLE_ID)
      AND NOT EXISTS (SELECT 1 FROM MOTORCYCLE m
        WHERE m.MOTORCYCLE_ID = :new.VEHICLE_ID))
    OR (:new.DISCRIMINATOR = 'Truck'
      AND EXISTS (SELECT 1 FROM TRUCK t WHERE ...)
      AND NOT EXISTS (SELECT 1 FROM CAR c WHERE ...)
      AND NOT EXISTS (SELECT 1 FROM MOTORCYCLE m WHERE ...))
    OR (:new.DISCRIMINATOR = 'Car' AND ...)
    OR (:new.DISCRIMINATOR = 'Motorcycle' AND ...));

  IF l_count = 0 THEN raise_application_error
      (-20101, 'OCL constraint GS_Vehicle_Types violated!');
  END IF;
END;
```

the paper, the authors discuss possible ways to realize constraints for 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 meta-properties of generalization sets in OntoUML have the same structure – they are based on multiplicities of related records 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.

6.2. Transformation of UML PIM into RDB PSM

As discussed in subsection 5.2, generalization sets can be transformed into a *single table*, *individual tables* or *related tables*. Each of them has advantages and disadvantages.

The *single table* realization is suitable in situations when we transform subclasses with few of attributes. Otherwise, the constraints for the mandatory attributes of the subclasses are getting complicated. Also, this solution is suitable in the case of `overlapping` generalization sets, as all the data are stored in a single table, preventing data and structure duplicities.

In contrast to that, the *related tables* realization is suitable in situations when the subclasses have many attributes, as the realization of their multiplicity constraints is much easier. On the other hand, it requires joining data from multiple tables to query data of instances of the subclasses. Moreover, it is more complicated to preserve the meta-properties of the generalization set, as queries into other tables will be needed in the validation checks, which increases the execution time of the queries and DML operations.

We consider the possible realization by the *individual tables* not suitable generally, as it leads to structural duplicities and data distribution, which is even more obvious in the case of `overlapping` and `incomplete` generalization sets.

6.3. Transformation of RDB PSM into SQL ISM

As discussed in subsection 5.3, the OCL constraints defined as the result of the transformation of the UML PIM into the RDB PSM can be realized by *database views*, *CHECK constraints* and *triggers*.

The *database views* can be used to query only valid data from the database. The *updatable database views* can be even used to manipulate with the data while checking the constraints. However, still, it is possible to create invalid data in the database violating the constraints by using the tables directly. Therefore, the responsibility is transferred to the application level.

The realization by the *CHECK constraints* is able to ensure the data consistency and prevent creating invalid data violating the constraints. However, as current common database engines do not support subqueries in the CHECK constraint statements, they can be used only in the case of the generalization sets realized by the *single table* approach.

The most reliable and universal realization is the realization by the *triggers*. Using the triggers, it is possible to completely validate the manipulated data and entirely prevent creating data violating the constraints. On the other hand, the triggers increase the time to execute the DML operations. However, as discussed in [32], this time increase is not substantial, unless manipulating with very large database.

6.4. Attribute Multiplicity

In this paper, we focused on the most common situation of mandatory attributes (attribute multiplicity `[1..1]`), as in OntoUML, optional attributes (minimal multiplicity `0`) are considered anti-patterns and they typically signal missing anti-rigid types like Roles or Phases [33]. Still, our approach is applicable even for the case of optional attributes. Some of the constraints will even simplify – e.g. the NOT NULL constraints for individual columns representing the attributes of the subclasses (Constraint 1).

Also, the collection attributes (attributes with the maximal multiplicity $\star$) can be simply realized by our approach. They just need to be transformed into separate classes and *one-to-many* relations, leading to the realization in the form of references and FOREIGN KEY constraints.

7. Conclusions

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

In the transformations, various options are available and additional constraints should be defined and realized to preserve 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 research questions formulated in section 1:

– Are there benefits of using OntoUML Kinds and Subkinds in PIM? — Yes, when following the proposed transformation, the resulting RDB is able to ensure the constraints, thus enabling better alignment with the problem domain.
– Is it possible to preserve generalization sets constraints of Kinds- and Subkinds in a relational database? — Yes, our transformation covered them.

OntoUML specifies numerous entity types and relation types. There are relatively a lot of rules and constraints (compared to e.g. UML) posed on them. As for the future research, a similar work as presented here should be elaborated for other important OntoUML constructs – the Non-sortal universal types – e.g. Category, Mixin, RoleMixin – and relational constructs – part-whole relations, Relators, etc. As for the continuation of the presented method, 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, mostly with the respect to quantitative measures concerning time and space efficiency in various database backends. Last but not least, a (semi)automated tooling needs to be developed to be able to use the method efficiently in the MDD lifecycle.

Acknowledgments. This research was supported by CTU grant No. SGS17/211/OHK3/3T/18 and contributes to the CTU's ELIXIR CZ Service provision plan.

References

1. Aleksić, S., Ristić, S., Luković, I.: An approach to generating server implementation of the inverse referential integrity constraints. In: Proceedings. AL-Zaytoonah University of Jordan, Amman, Jordan (May 2011)
2. Arlow, J., Neustadt, I.: UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition). Addison-Wesley Professional (2005)
3. Barcelos, P.P.F., Guizzardi, G., Garcia, A.S., Monteiro, M.: Ontological evaluation of the ITU-T recommendation g.805. vol. 18. IEEE Press, Cyprus (2011)
4. Benevides, A.B., Guizzardi, G., Braga, B.F.B., Almeida, J.P.A.: Assessing Modal Aspects of OntoUML Conceptual Models in Alloy. In: Advances in Conceptual Modeling - Challenging Perspectives, vol. 5833, pp. 55–64. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), DOI: 10.1007/978-3-642-04947-7_8
5. Dimitrieski, V., elikovic, M., Aleksic, S., Risti, S., Alargt, A., Lukovi, I.: Concepts and evaluation of the extended entity-relationship approach to database design in a multi-paradigm information system modeling tool. Computer Languages, Systems & Structures 44, 299–318 (Dec 2015)
6. DresdenOCL 3.4.0. `https://github.com/dresden-ocl/dresdenocl` (Aug 2014), accessed: 2016-02-11

7. Egea, M., Dania, C.: SQL-PL4ocl: an automatic code generator from OCL to SQL procedural language. Software & Systems Modeling pp. 1–23 (May 2017), `https://link.springer.com/article/10.1007/s10270-017-0597-6`
8. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edn. (2002)
9. Goncalves, B., Guizzardi, G., Pereira Filho, J.G.: Using an ECG reference ontology for semantic interoperability of ECG data. Special Issue on Ontologies for Clinical and Translational Research (2011)
10. Guizzardi, G.: Ontological Foundations for Structural Conceptual Models, vol. 015. University of Twente, Enschede (2005)
11. Guizzardi, G.: Agent roles, qua individuals and the counting problem. Software Engineering of Multi-Agent Systems (IV) (2006)
12. Guizzardi, G.: The Problem of Transitivity of Part-Whole Relations in Conceptual Modeling Revisited. In: Proccedings of 21st International Conference on Advanced Information Systems Engineering (CAISE09). Amsterdam, The Netherlands (2009), 00000
13. Guizzardi, G.: Ontological Meta-properties of Derived Object Types. In: Advanced Information Systems Engineering, pp. 318–333. No. 7328 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jun 2012), `http://link.springer.com/chapter/10.1007/978-3-642-31095-9_21`, dOI: 10.1007/978-3-642-31095-9_21
14. Guizzardi, G., Wagner, G.: A unified foundational ontology and some applications of it in business modeling. In: In CAiSE Workshops (3. pp. 129–143 (2004)
15. Guizzardi, G., Wagner, G., Guarino, N., Sinderen, M.v.: An Ontologically Well-Founded Profile for UML Conceptual Models. In: Advanced Information Systems Engineering, pp. 112–126. No. 3084 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jun 2004), `http://dx.doi.org/10.1007/978-3-540-25975-6_10`
16. Heidenreich, F., Wende, C., Demuth, B.: A Framework for Generating Query Language Code from OCL Invariants. Electronic Communications of the EASST 9(0) (Nov 2007), `https://journal.ub.tu-berlin.de/eceasst/article/view/108`
17. Luković, I., Mogin, P., Pavićević, J., Ristić, S.: An approach to developing complex database schemas using form types. Software: Practice and Experience 37(15), 16211656 (Dec 2007), `http://dx.doi.org/10.1002/spe.v37:15`
18. Luković, I., Popović, A., Mostić, J., Ristić, S.: A tool for modeling form type check constraints and complex functionalities of business applications. Computer Science and Information Systems 7(2), 359–385 (2010)
19. Martin, R.C.: Design principles and design patterns. Object Mentor 1, 34 (2000)
20. Mellor, S.J., Clark, A.N., Futagami, T.: Model-driven development. IEEE Software 20(5), 14 (Sep 2003)
21. Melton, J.: Advanced SQL:1999. Morgan Kaufmann Publishers (2003)
22. Mok, W.Y., Paper, D.P.: On transformations from UML models to object-relational databases. In: Proceedings of the 34th Annual Hawaii International Conference on System Sciences. pp. 10 pp.– (Jan 2001)
23. Obrenović, N., Popović, A., Aleksić, S., Luković, I.: Transformations of check constraint PIM specifications. Computing and Informatics 31(5), 1045–1079 (2012)
24. OMG: Object constraint language (OCL), version 2.4. `http://www.omg.org/spec/OCL/2.4/` (Feb 2014), accessed: 2016-02-23
25. OMG: UML 2.5. `http://www.omg.org/spec/UML/2.5/` (Mar 2015), accessed: 2016-02-08
26. Oracle: Oracle SQL. `http://www.oracle.com/technetwork/database/database-technologies/sql/overview/index.html`, accessed: 2017-01-19
27. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd Edition. McGraw-Hill, Boston, 3rd edition edn. (Aug 2002)

28. Rybola, Z., Pergl, R.: Towards OntoUML for Software Engineering: Introduction to the Transformation of OntoUML into Relational Databases. In: Enterprise and Organizational Modeling and Simulation. LNBIP, Springer, CAiSE 2016, Ljubljana, Slovenia (June 2016), in press.
29. Rybola, Z., Pergl, R.: Towards OntoUML for Software Engineering: Transformation of Anti-Rigid Sortal Types into Relational Databases. In: Model and Data Engineering. LNCS, vol. 9893, pp. 1–15. Springer, MEDI 2016, Almria, Spain (Sep 2016)
30. Rybola, Z., Pergl, R.: Towards OntoUML for Software Engineering: Transformation of Rigid Sortal Types into Relational Databases. In: Proceedings of the 2016 Federated Conference on Computer Science and Information Systems. p. 15811591. No. 8 in ACSIS, Gdansk, Poland (2016), doi: 10.15439/2016F250
31. Rybola, Z., Pergl, R.: Transformation of Kinds and Subkinds into Relational Databases: A Running Example. Technical Report TR-FIT-2017, Czech Technical University in Prague (2017)
32. Rybola, Z., Richta, K.: Possible Realizations of Multiplicity Constraints. Computer Science and Information Systems 10(4), 1621–1646 (Oct 2013), wOS:000327912000006
33. Sales, T.P., Guizzardi, G.: Anti-patterns in Ontology-driven Conceptual Modeling: The Case of Role Modeling in OntoUML. In: Ontology Engineering with Ontology Design Patterns: Foundations and Applications. IOS Press (2016)
34. da Silva, A.R.: Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures 43, 139 – 155 (2015)
35. Sparks, G.: Database Modeling in UML, `http://www.eetimes.com/document.asp?doc_id=1255046`, accessed: 2016-02-02
36. Sparx Systems: Enterprise architect 13. `http://www.sparxsystems.com.au/products/ea/index.html`, accessed: 2017-01-02
37. Torres, A., Galante, R., Pimenta, M.S., Martins, A.J.B.: Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. Information and Software Technology 82, 1–18 (Feb 2017)
38. Vara, J.M., Vela, B., Bollati, V.A., Marcos, E.: Supporting Model-Driven Development of Object-Relational Database Schemas: A Case Study. In: Theory and Practice of Model Transformations, vol. 5563, pp. 181–196. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), dOI: 10.1007/978-3-642-02408-5_13
39. Zamborlini, V., Guizzardi, G.: On the representation of temporally changing information in OWL. In: Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2010 14th IEEE International. pp. 283–292. IEEE (2010)

Zdeněk Rybola is an assistant professor at the Department of Software Engineering at the Faculty of Information Technology, Czech Technical University in Prague, teaching software engineering courses. He also just submitted his PhD thesis in August 2017. His area of interest includes OntoUML, UML, Model-Driven Development and Relational databases.

Robert Pergl is an assistant professor at the Department of Software Engineering at the Faculty of Information Technology, Czech Technical University in Prague and the head of the research group Centre for Conceptual Modelling and Implementation (CCMi). He focuses on ontologies, conceptual modelling, enterprise engineering, programming languages and paradigms.

Received: January 9, 2017; Accepted: August 17, 2017.