Transformation of Ontology-Based Conceptual Models into Relational Schemas

Gustavo L. Guidoni[1,2], João Paulo A. Almeida[1], and Giancarlo Guizzardi[1,3]

[1] Ontology & Conceptual Modeling Research Group (NEMO),
Federal University of Espírito Santo, Vitória, Brazil
[2] Federal Institute of Espírito Santo, Colatina, Brazil
[3] Free University of Bozen-Bolzano, Italy
gustavo.guidoni@ifes.edu.br, jpalmeida@ieee.org, gguizzardi@unibz.it

Abstract. Despite the existence of several strategies for transforming structural conceptual models into relational schemas, there are a number of features of ontology-based conceptual models that have not been taken into account in the existing literature. Most approaches fail to support conceptual models that: (i) include overlapping or incomplete generalizations; (ii) support dynamic classification; (iii) have multiple inheritance; and (iv) have orthogonal hierarchies. This is because many of the approaches discussed in the literature are based on the object-relational mapping and, as a consequence, assume primitives underlying object-oriented programming languages (instead of conceptual modeling languages). This paper addresses this gap, focusing on the realization of taxonomic hierarchies of ontology-based conceptual models. We explore some ontological meta-properties that characterize classes in these models (sortality and rigidity) to guide the transformation and avoid some problems in existing approaches.

Keywords: Object-relational mapping · transformation · impedance mismatch · ontology primitives.

1 Introduction

Conceptual models play an important role in the design of relational databases, and are often used to guide the definition of relational schemas. Several systematic model transformation approaches to this end have been explored in the academic literature and incorporated in production-ready tools [23]. In these approaches, elements and patterns of a resulting relational schema have their origin traced back to corresponding elements and patterns of a source conceptual model. By using a model transformation approach, design decisions are incorporated into transformation specifications; automated model transformation then shields designers from manual (error-prone) realization steps.

A significant challenge of a model transformation approach is to preserve the semantics of a source model. This is because, often, source and target models are based on different paradigms, employ different concepts, which results in a variety of technical problems. A manifest example of this is the so-called "Object-Relational Impedance Mismatch" [15], which results from a "semantic gap" with object-oriented constructs not bearing a direct correspondence with constructs in relational schemas.

The existing transformation approaches that target relational schemas vary in a number of ways, including: (i) the primitives with which the source conceptual model is defined (e.g., as given by the source modeling language and its underlying abstractions), (ii) the realization strategies employed to bridge the semantic gap, (iii) the non-functional properties of the resulting database systems (such as time performance, ease of use, maintainability), and (iv) level of automation.

Consider, for example, approaches to transform an object-oriented inheritance hierarchy into relational schemas (such as those discussed by [2, 16]). Concerning the adopted primitives (i), their vast majority assume objects are classified statically (i.e., objects cannot change classes at runtime); some of them assume single inheritance only. Concerning the realization strategies employed (ii), approaches adopt variations of *one table per class*, *one table per leaf class* and *one table per hierarchy*. Strategy choices are either fixed in a particular approach or discussed with general heuristics. For example, when discussing these approaches, Ambler [2] argues that if priority is given to the support for polymorphism, then the best strategy is *one table per class*, at the cost of performance. Keller [16] indicates the use of *one table per hierarchy* strategy if the transformation purpose is performance and maintainability.

Despite the existence of several strategies for transforming object-oriented models into relational schemas, there are a number of features of ontology-based conceptual models that have not been taken into account in the existing literature. Most approaches do not cater for source conceptual models that: (i) include overlapping or incomplete generalizations; (ii) support dynamic classification; (iii) have multiple inheritance; and (iv) have orthogonal hierarchies. This is because many of the approaches discussed in the literature are based on the object-relational mapping and, as a consequence, assume primitives underlying object-oriented programming languages (instead of conceptual modeling languages). This paper addresses this gap, focusing on the realization of taxonomic hierarchies of ontology-based conceptual models, which do not adhere to the constraints of inheritance hierarchies in programming languages. Further, by exploring ontological distinctions for types—specifically the formal metaproperties of *sortality* and *rigidity*—we are able to devise a novel transformation strategy and avoid some problems in existing approaches.

This paper is further structured as follows. Section 2 presents the primitives we assume in a source conceptual model. It also introduces a running example. Section 3 identifies predominant strategies in the literature to transform class hierarchies into relational schemas; it identifies limitations that motivate us to investigate a novel approach. Section 4 presents the ontology-based approach, which is applied to the running example. Section 5 discusses how the proposed approach is positioned with respect to the dominant strategies and other related work. Section 6 presents concluding remarks.

2 Primitives of the Source Conceptual Model

We assume that the basic elements of a taxonomy in a structural conceptual model are *classes* and their relations of *specialization* (also called "is-a", *subclassing*, or *inheritance* relations). Classes are used to capture common properties of entities they classify, and, in a taxonomic hierarchy, more general classes are specialized into more

specific (sub-)classes, which "inherit" attributes and associations of their superclasses (for brevity, we call here both the attributes and associations of a class its "features"). We assume conceptual modelling approaches share these ground notions, nevertheless, there are variations including additional supporting mechanisms, their semantics and their possible range of use, as discussed in the remainder of this section.

Multiple Inheritance. A first source of variation concerns the possibility of a subclass to specialize more than one superclass. In a taxonomic hierarchy with multiple inheritance, a class can be a subclass of different classes [6]. A subclass in such a hierarchy inherits the properties of all its superclasses. Multiple inheritance has been avoided in some programming languages as it leads to some implementation difficulties. In conceptual modeling, however, multiple inheritance is hardly dispensable, as it enables opportunities for modularity and reusability [7].

Overlapping Classification. Another variation concerns whether an object can simultaneously instantiate multiple classes which are not related by specialization. For example, a person may instantiate both the `BrazilianCitizen` and the `ItalianCitizen` subclasses of `Person`. In UML, this can be explicitly supported with the so-called *overlapping generalization sets*, in which a set of non-disjoint classes specialize the same superclass. Additionally, this kind of scenario can be supported with different–orthogonal–hierarchies that specialize a common superclass based on different criteria. For example, persons may be classified according to their age and according to citizenship status. In this setting, a Brazilian adult would instantiate both the `BrazilianCitizen` and the `Adult` subclasses of `Person` (each from a different generalization set).

Non-Exhaustive Classification. A related variation concerns whether specializing subclasses "cover" the specialized superclass, i.e., whether they jointly exhaust all the classification possibilities for the superclass. In UML, this can be explicitly supported with the so-called *complete* generalization sets, which are opposed to *incomplete* generalization sets. In the case of an incomplete generalization set, it is possible for an instance of the superclass not to instantiate any of the subclasses in the set. For example, it is possible for a person to be stateless (in the sense of not being considered a national by any State), and hence a nationality generalization set could be marked as "incomplete" even in the case all known nationalities are explicitly modeled.

Dynamic Classification. Another variation we consider concerns whether instances can change the set of classes they instantiate throughout their existence. For example, a `Person` may be reclassified from `Child` to `Adult` with the passing of time. This is not possible if static classification is assumed. Many modeling languages support only static classification given their roots in object-oriented programming languages that likewise only support static classification; in these languages, the class that an object instantiates is defined at object instantiation time, and remains fixed throughout that object's life cycle. Nevertheless, in conceptual modeling, dynamic classification has been considered an important feature and studied by several authors [1, 9, 21, 22, 25]. Dynamic classification enlarges the realm of classes to include those which apply contingently or temporarily to their instances. Examples include the ontological notions of *phases* (such as `Child` and `Adult`), and *roles* (such as `BrazilianCitizen`, `ItalianCitizen`, `Employee` and `Customer`).

Abstract and Concrete Classes. Finally, we assume that the conceptual modeling technique may distinguish between abstract and concrete classes. Abstract classes have no "direct" instances, i.e., all of their instances are also instances of specializing subclasses. Concrete classes in their turn are not bound by this constraint (and thus can have "direct" instances).

Running Example. Figure 1 shows a UML model exploring all of the aspects of a source conceptual model we address in this paper, and is used further as a running example. It includes: (i) an overlapping and incomplete generalization set, in which `Persons` are specialized according to—none or more than one—enumerated countries of citizenship; (ii) a generalization set orthogonal to the first one, in which `Persons` are classified dynamically according to life phase; (iii) multiple inheritance, with each `PersonalCustomer` being both a `Customer` and an `Adult Person`, as well as each `CorporateCustomer` being both a `Customer` and an `Organization`); (iv) orthogonal classification hierarchies (with `Organization` being classified as a `CorporateCustomer` when it establishes a relation with another `Organization` and also possibly being classified as a `PrimarySchool` in which children may be enrolled or as a `Hospital`); (v) an abstract class `NamedEntity`, which is specialized into `Person` and `Organization` and an abstract class `Customer`, which is specialized into concrete classes `PersonalCustomer` and `CorporateCustomer`.

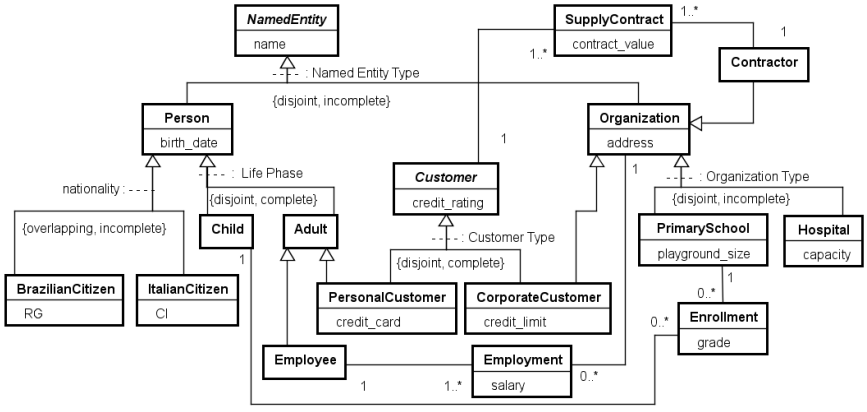


Fig. 1. Running example

3 Current Realization Strategies

The relational model does not directly support the concept of inheritance, and, hence, realization strategies are required to preserve the semantics of a source conceptual model in a target relational schema. Such strategies are described by several authors [3, 8, 16, 18, 23] under various names. In this section, we review the most salient strategies in the literature. We discuss their applicability in relation to the source conceptual modeling primitives under discussion.

One table per class. This strategy is also called "Class-table" [8], "Vertical inheritance" [23] or "One class one table" [16]: In this strategy, each class gives rise to a separate table, with columns corresponding to the class's features. In this strategy, specialization between classes in the conceptual model gives rise to a foreign key in the table that corresponds to the subclass (henceforth "subclass tables" for simplicity). This foreign key references the primary key of the table corresponding to the superclass (henceforth "superclass table" for simplicity). For example, when applying this strategy under the hierarchy formed by the classes `Customer`, `PersonalCustomer` and `CorporateCustomer` in Figure 1, all classes are transformed into tables. Foreign keys in the `PERSONAL_CUSTOMER` and `CORPORATE_CUSTOMER` tables refer to the primary key of the `CUSTOMER` table. The relational schema directly reflects the organization of classes in the conceptual model, and no restriction on the primitives of the model are imposed. Multiple inheritance can be supported by using a composite foreign key in subclass tables (`PERSONAL_CUSTOMER` in fact has a composite key referencing the primary keys of the `ADULT` and `CUSTOMER` tables). Constraints on a generalization set (overlapping and non-exhaustive classification) are reflected in integrity constraints concerning the cardinality of entries in the subclass tables for a particular row in each superclass table. Dynamic classification is implemented by deletion of a row in a subclass table (cascaded to further subclass tables) and, possibly, insertion in another. Abstract classes and concrete classes are treated alike. The main drawback of this approach is its performance characteristics. In order to manipulate a single instance of a class, e.g., to read all its attributes or to insert a new instance with its attributes, one needs to traverse a number of tables corresponding to the depth of the whole specialization hierarchy. For example, consulting the `name` and `credit_card` of a `Person` one needs to traverse four(!) tables, namely `NAMED_ENTITY`, `PERSON`, `ADULT` and `PERSONAL_CUSTOMER` (and even more tables if we are also interested in a person's nationality). This limitation motivates the adoption of other strategies, as discussed in the sequel.

One table per leaf class. In this strategy, also termed "horizontal inheritance" [23], each of the leaf classes in the hierarchy gives rise to a corresponding table. Features of all (non-leaf) superclasses of a leaf class are realized as columns in the leaf class table. No foreign keys emulating inheritance are employed in this approach. The strategy can be understood as reiterated application of an operation of "flattening" of superclasses. For example, when applying this strategy under the hierarchy formed by the classes `Customer`, `PersonalCustomer` and `CorporateCustomer` in Figure 1, the `credit_rating` attribute of the `Customer` correspond to columns in both the `PERSONAL_CUSTOMER` and `CORPORATE_CUSTOMER` tables (which also has columns for attributes of its other superclasses: `birth_date` and `address` respectivelty). Any references to a superclass (e.g., the references realized as foreign keys of a `SUPPLY-CONTRACT`) now refer to an entry in either of the subclass tables (in this case `PERSONAL_CUSTOMER` or `CORPORATE_CUSTOMER`). Referential integrity tends to be problematic if the number of leaf classes is large. For example, if there are two references from a superclass to another class in the model and ten leaf classes, there will be the need to maintain referential integrity for twenty references. Special attention is required when the conceptual model has multiple inheritance, because of possible name collision (easily addressed with name conventions). (If multiple inheritance is disallowed,

the strategy becomes equivalent to *one table per inheritance path*.) This strategy addresses the performance issue discussed for the *one table per class*, at the cost of polymorphic queries. Consider, for example, a query for all customers to ascertain average `credit_rating`. In the *one table per class* strategy such a query involves only one table. In this strategy, however, the query requires the union of `PERSONAL_CUSTOMER` and `CORPORATE_CUSTOMER`). The higher the class in the specialization hierarchy, the higher the number of classes involved in a polymorphic query. Regardless of the performance characteristics, there is a more serious concern when it comes to supporting overlapping generalization sets and orthogonal hierarchies. In the presence of overlapping classification, there is an issue with the identification of an instance of the superclass. Consider the "flattening" of `Person` in our example. In case a person has double Brazilian and Italian citizenship, there would be a row in the `BRAZILIAN_CITIZEN` table and another row in the `ITALIAN_CITIZEN` table denoting the same person, but without a correlating identifier. This problem is also present for orthogonal hierarchies of an abstract superclass. (There would be, e.g., a row in the `ADULT` table corresponding to a row in the `BRAZILIAN_CITIZEN` table.) Further, there is a problem with the preservation of identity in dynamic classification (when objects change classes, a row is deleted from one table and added to another, all attributes that are inherited from superclasses must be copied to the new row). Differently from *one table per class*, there is no stable identifier.

One table per hierarchy. This strategy, also called "Single-table" [8] or "One inheritance tree one table" [16], can be understood as the opposite of *one table per leaf class*, applying a "lifting" operation to subclasses instead of the "flattening" of superclasses. Consider, e.g., the hierarchy formed by `Customer`, `PersonalCustomer` and `CorporateCustomer`. `Customer` is the top-level class in this hierarchy, and will thus give rise to a corresponding `CUSTOMER` table. Attributes of each subclass become columns in the superclass table, with mandatory attributes corresponding to optional columns. This strategy usually requires the creation of an additional column to distinguish which subclass is (or which subclasses are) instantiated by the entity represented in the row (a so-called "discriminator" column). The "lifting"operation is reiterated until the top-level class of each hierarchy is reached. In principle, the performance problems discussed for the other strategies do not appear in this approach. However, as discussed in [23], standard database integrity mechanisms cannot prevent certain inconsistencies. In our example, the `ENROLLMENT` table would have foreign keys to the top-level class `NAMED_ENTITY`, since `Person` and `Primary School` would be "lifted" to the corresponding top-level class. Thus, the discriminator would have to be checked to make sure that only children are enrolled in primary schools, because the database would admit any named entity enrolled in another named entity, e.g., hospitals enrolled in hospitals. In addition, the greater the number of leaf classes in a hierarchy, the greater the number of optional columns that remain unattributed in every row. This approach is problematic in the face of multiple inheritance. If multiple inheritance is admitted, then there may be top-level classes that are not disjoint (e.g., `Customer` and `NamedEntity`). This means that there will be rows in more than one table denoting the same individual, a problem which also appeared in *one table per leaf class*, albeit for different reasons. Dynamic classification at the top of the hierarchy (such as the case of `Customer`) also poses a challenge, not unlike the one faced by *one table per leaf class*.

4 Ontological Semantics to the Rescue

In the previous section, we have observed that there are a number of deficiencies of existing conceptual model transformation approaches, with (i) poor performance in various data manipulation operations, (ii) failure to explore beneficial database mechanisms, and/or (iii) lack of support for various conceptual modeling primitives including orthogonal classification hierarchies, overlapping non-exhaustive generalization sets as well as dynamic classification and multiple inheritance.

In contrast with all the aforementioned approaches, our proposal in this paper explores the *ontological semantics* [12] of the elements represented in a conceptual model. By identifying formal ontological meta-properties of the classes in a model, including *sortality* and *rigidity*, we are able to guide the transformation. We use here the ontological distinction underlying the Unified Foundational Ontology (UFO) [12], which have their roots in OntoClean [11]. Here we discuss only the ontological distinctions that are needed in this paper. For further reference and formalization, see [12, 14].

Take a subject domain focused on objects (as opposed to events or occurrences). Central to this domain we will have a number of object *kinds*, i.e., the genuine fundamental types of objects that exist in this domain. The term "kind" is meant here in a strong technical sense, i.e., by a kind, we mean a type capturing essential properties of the things it classifies. In other words, the objects classified by that kind could not possibly exist without being of that specific kind. In Figure 1, we have represented two object kinds, namely, `Person` and `Organization`. These are the fundamental kinds of entities that are deemed to exist in the domain. Kinds tessellate the possible space of objects in that domain, i.e., all objects belong necessarily to exactly one kind.

Static subdivisions (or subtypes) of a kind are naturally termed *subkinds*. In our example, the kind `Organization` is specialized in the subkinds `Primary School` and `Hospital`. Object kinds and subkinds represent essential properties of objects, i.e., properties that these objects instantiate in all possible situations. They are examples of what are termed *rigid* or static types. There are, however, also types that represent contingent or accidental properties of objects (termed *anti-rigid* types). These include *phases* and *roles*. Phases represent properties that are intrinsic to entities; roles, in contrast, represent properties that entities have in a relational context, i.e., contingent relational properties. In our example, we have a *phase partition* including `Child` and `Adult` (as phases in the life of a `Person`). Several other types in the example are *roles*: `Employee`, `Contractor`, `BrazilianCitizen` and `ItalianCitizen` (the last two in the context of a relation with a national state, not represented in the model, for simplicity).

Kinds, subkinds, phases, and roles are all object *sortals*. In the philosophical literature, a sortal is a type that provides a uniform principle of identity, persistence, and individuation for its instances. A sortal is either a kind (e.g., `Person`) or a specialization of a kind (e.g., `Child`, `Employee`, `Hospital`), i.e., it is either a type representing the essence of what things are or a sub-classification of entities that "have that same type of essence". There are also types that apply to entities of multiple kinds, these are called *non-sortals*. An example of non-sortal is `Customer` (which can be played by both people and organizations). We call these role-like types that classify entities of multiple kinds *role mixins*. Another example of non-sortal is `NamedEntity`. However, it is a rigid non-sortal, classifying objects of various kinds statically.

In addition to objects, there are also *existentially dependent* endurants, i.e., endurants that depend on other endurants for their existence. Here, we highlight the so-called *relators*, which reify a relationship *mediating* endurants. In our example, an instance of `Employment` can only exist as long as a particular instance of `Person` (playing the `Employee` role) and a particular instance of `Organization` (playing the corresponding `Employer` role, omitted here) exist. The meta-properties we have discussed for object types also apply to relator types. In our example, `Employment`, `Enrollment` and `SupplyContract` are relator kinds.[4]. Relators are composed of another type of dependent endurant termed a *qua-entity* (or *role instance*) [12]. Each qua-entity composing a relator inheres in one of the relatum of that relator while being relationally dependent on the other relata.

Figure 2 revisits Figure 1, now including class stereotypes according to the ontological distinctions discussed above, which are part of UFO-based OntoUML profile [12]. According to the rules that apply to OntoUML (formally characterized in [14]):

- non-sortals (such as ≪category≫ and ≪roleMixin≫), when present in a model, are always superclasses (and never subclasses) of sortals (such as ≪kind≫, ≪subkind≫, ≪role≫, ≪phase≫, ≪relatorKind≫);
- non-sortals are abstract and are only instantiated through their sortal subclasses;
- sortals that are not kinds (≪subkind≫, ≪role≫, ≪phase≫ or ≪role≫) specialize *exactly one* `kind` (or ≪relatorKind≫), from which they inherit their principle of identity. So, there is no multiple inheritance of kinds, since all kinds are mutually disjoint;
- rigid types (≪kind≫, ≪subkind≫, ≪category≫, ≪relatorKind≫) never specialize anti-rigid types (≪role≫, ≪roleMixin≫ or ≪phase≫).

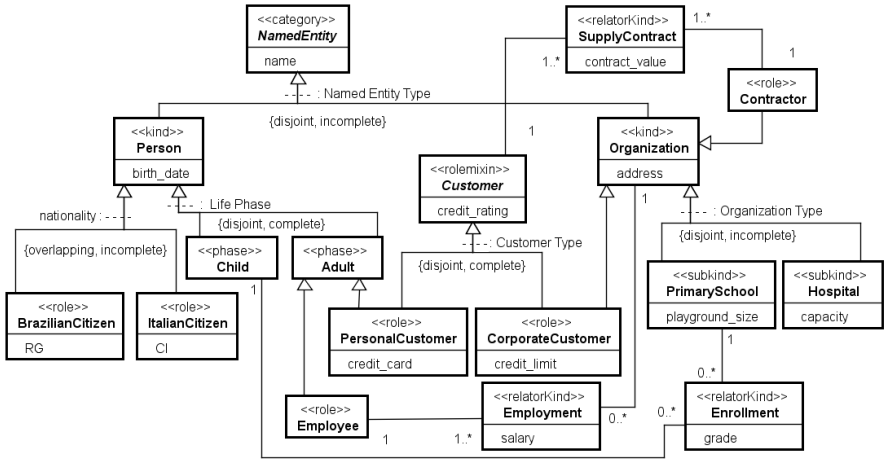


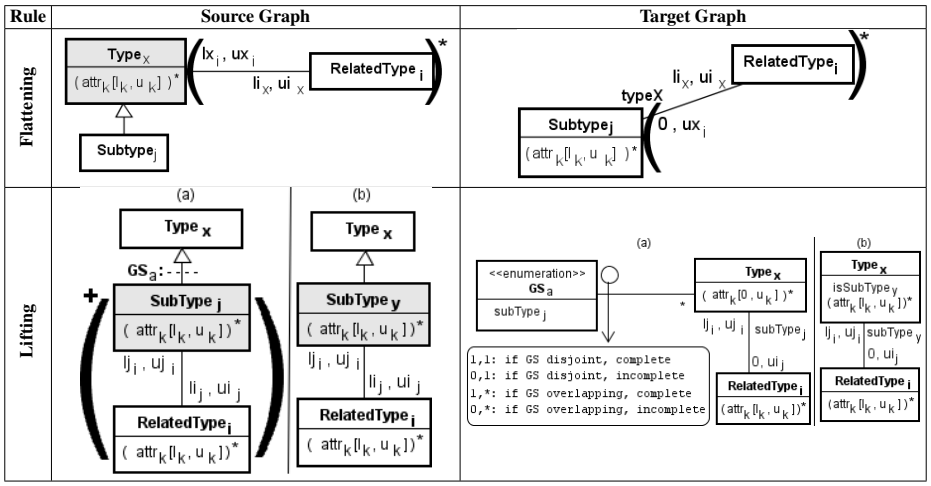
Fig. 2. Running example with OntoUML stereotypes added.

[4] It is not in the scope of this paper to discuss strategies for the representation of *n*-to-*n* relationships. With the reification of these relationships into relators, the challenge is already addressed at the conceptual model level, with many other benefits [10].

We can now use the ontological distinctions to articulate a transformation strategy. In a nutshell, this strategy results in a schema composed of tables corresponding to the *kinds* of entities in the domain. Because of this, it is termed *one table per kind*.

The first two steps of our approach are based on the operations of *flattening* and *lifting*, which are guided by the aforementioned ontological distinctions. Non-sortals are flattened towards sortals (step 1). Sortals are lifted until their kinds are reached (step 2). These operations basically correspond to the graph transformation *model abstraction rules* proposed in [13]. In particular, the former to the *non-sortal abstraction rule* (R_2), and the latter to a combination of the *sortal abstraction rule* (R_3) and the *subkind and phase partition abstraction rule* (R_4). Flattening is performed from all top-level non-sortals. In our running example, the `name` attribute `NamedEntity` is flattened to `PERSON` and `ORGANIZATION`. The same applies to `credit_rating` in `Customer`. When all non-sortals have been flattened, lifting is performed recursively from the leaves of the inheritance tree, propagating mandatory attributes as optional, until kinds are reached.[5] Table 1 shows the two operations as graph transformations. The classes flattened or lifted are shown in grey. For lifting, there are two cases. When a generalization set is involved (a), a discriminator enumeration is introduced, with labels corresponding to each $SubType_j$ in the set. Otherwise (b), a Boolean attribute suffices.

Table 1. Transformation Patterns.



After these operations have been carried out, as a final step of our approach (step 3), tables are produced for each of the classes in the refactored model. The tables corresponding to dependent entities must have foreign keys to the entities on which they depend. This is the case for tables corresponding to relator kinds, and also for the discriminating tables in the case of overlapping generalization sets. In the latter case, each row in a discriminator table represents a qua-entity connecting role players with the

[5] For the implementation repository see `https://github.com/nemo-ufes/ontouml2db`

corresponding (reified) role class. As previously discussed, qua-entities and relators are existentially dependent entities.

Figure 3 presents the schema that results from the application of these transformation steps in the conceptual model in Figure 2. We obtain the five tables corresponding to object kinds: `PERSON`, `ORGANIZATION`, and three corresponding to relator kinds: `EMPLOYMENT`, `ENROLLMENT` and `SUPPLY_CONTRACT`. An additional table for the discriminator that results from the overlapping generalization set `nationality` is introduced (`PERSON-NATIONALITY`, representing a qua-entity connecting a person to a particular nationality type). Finally, for all the tables representing dependent entities types, we introduce the corresponding dependency keys.

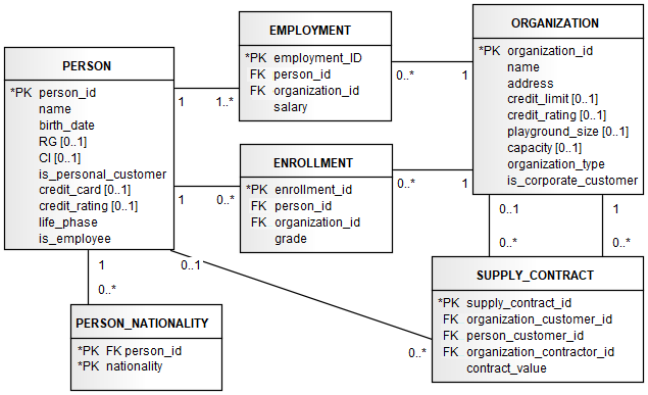


Fig. 3. Resulting relational schema in running example *one table per kind*.

5 Discussion and Comparison to Alternative Approaches

Table 2 summarizes the comparison between the proposed *one table per kind* strategy and the three dominant strategies in the literature, where: n is the total number of classes in the source conceptual model, h is the maximum height of the hierarchy (i.e., maximum path size from a top-level class to a leaf class), n_l is the number of leaf classes in the hierarchy, n_t the number of top-level classes, and n_k is the number of kinds. Note that the number of kinds (n_k) is equal to or lower than the number of leaf classes (i.e., $n_k \leq n_l \leq n$), and that they are equal ($n_k = n_l$) only in case there are no subkinds, roles and phases. Thus, the number of tables to required to represent entities in the domain in the proposed *one table per kind* strategy is equal to or lower than that required by *one table per class* and *one table per leaf class*. The comparison with *one table per hierarchy* requires us to consider the number of top-level classes (n_t). The two approaches result in the same number of tables when there are no non-sortals ($n_k = n_t$).

The table also presents worst-case figures for the retrieval and insertion of an entity (with all its attributes). *One table per class* fares poorly in this comparison, with h joins

Table 2. Comparison between Realization Strategies.

Realization Strategy	Nº of tables representing entities	Nº of joins to retrieve an entity	Nº of tables affected in insert operation	Nº of tables in union to read one attribute (polymorphic query)	Multiple inheritance	Orthogonal hierarchies	Dynamic classification performance
One table per class	n	h	$h+1$	1	yes	yes	poor
One table per leaf class	n_l	1	1	n_l	yes	no	poor
One table per hierarchy	n_t	1	1	1	no	yes	good
One table per kind	n_k	1	1	n_k (1, if defined in sortal)	yes	yes	good

required in the worst case. The performance of polimorphic queries is considered, with respect to the number of tables involved in a union to read one attribute defined in a superclass. *One table per leaf class* performs poorly in this respect. All others perform equally, except *one table per kind* when the attribute is defined in a non-sortal, in which case, n_k unions may be required in the worst case (when the non-sortal class in which the attribute is defined classifies entities of all kinds in the model). Even in this case, the approach is equal to or better than *one table per leaf class* (since $n_k \leq n_l$). Table 3 shows the values for these variables for a number of models in different domains (those also employed in [13]), revealing that height of the hierarchy ranges from two to six, and the number of kinds in a model is typically one fourth or one fifth of the total number of classes. The average number of leaf classes (n_l) is 39, contrasted with 15 for kinds.

Problems with multiple inheritance in *one table per hierarchy* do not appear in *one table per kind* because there is no multiple inheritance of kinds. Multiple inheritance of non-kind sortals (subkinds, phases and roles) does not pose a problem, as discriminators identify the instantiated classes. Multiple inheritance of non-sortals creates no problems because they are flattened into kinds. Problems with orthogonal hierarchies and overlapping generalization sets in *one table per leaf class* also do not arise as a consequence of the transformation strategy. Kinds tables are where the entities primary keys are placed, and hence there is no problem with the same entity being represented in several tables. Flattening of non-sortals poses no problem in this scenario. In the lifting of non-kind sortals, orthogonal hierarchies and overlapping generalization sets are, not unlike multiple inheritance, reflected in discriminators in the kind table. Dynamic classification is supported naturally as reclassification is simply change in discriminators. This is not the case with *one table per class* and *one table per leaf class*, which require deletion and insertion, posing a problem for referential integrity.

In addition to the dominant strategies we have discussed, there are approaches which use the distinction between abstract and concrete classes, with impact on performance characteristics. For example, *one table per concrete class* is a variant of *one table per*

Table 3. Variable occurrences by OntoUML model.

Variables	**OntoUML Models**										
	Cloud Vulnerability	**ECG**	**G.805**	**MPOG**	**Normative Acts**	**OpenBio**	**OpenFlow**	**Open Provenance**	**PAS 77**	**Software Requirements**	**Average**
n	30	49	123	15	59	231	20	33	66	17	**64**
h	3	2	6	4	3	5	2	2	3	2	**3**
n_l	12	18	70	7	43	163	8	12	41	11	**39**
n_t	5	4	14	3	5	9	4	8	5	2	**6**
n_k	12	18	18	5	10	37	6	17	19	7	**15**

class in which abstract classes are flattened out. Since flattening of abstract classes reduces the height of the hierarchy, this strategy has the potential of improve the performance of retrieving and inserting an entity. Nevertheless, that performance is still much dependent on the size of the concrete class hierarchy. Further, dynamic classification performance remains a challenge in this approach. By identifying the ontological meta-properties of classes in the source conceptual model, we are able to better navigate performance tradeoffs, beyond what can be achieved with the abstract–concrete distinction. For example, strategies such as *one table per rigid sortal* become possible, approximating *one table per concrete class* in terms of performance but circumventing its difficulty with dynamic classification. This approach is favored by Rybola and Pergl [20], who focus on the transformation of sortals. In his Ph.D. thesis, Rybola [19] proposes the transformation of non-sortals with a pattern that introduces a table for each class. In this sense, his approach approximates *one table per class*, but produces even more tables due to the patterns employed to address the relation between the non-sortal and sortal hierarchies. This exacerbates the performance issues when accessing or inserting an object. The techniques proposed in that work can be adapted to our strategy. In particular, quite sophisticated integrity rules and validation triggers were proposed to preserve the semantics of the original constructs from the source OntoUML model.

Over the years, a number of authors have compared the various object-relational mapping strategies in terms of system infrastructure (performance and storage) and relational schema design (understanding and maintainability). Among these, Keller [16] points out the infrastructure and design "forces" that govern the development of a relational schema, as well as some characteristics used in the application design, such as polymorphism. The author also exposes the consequences of using the strategies, which also is done by Fowler [8] when identifying the strengths and weaknesses of each strategy. Ambler [2] performs a brief comparison between the strategies and is concerned with the practical differences between the relational and the object-oriented paradigms. Philippi [18] establishes the consequences of mapping strategies in relation to the infrastructure and design aspects of the relational schema when the inheritance hierarchy is associated with other classes of the model along with association cardinality. In turn, Torres [23] does not perform a systematic comparison between the strategies, but identifies their adoption in the various object-relational mapping tools. A common characteristic of all these efforts is their focus on the primitives of object-oriented programming languages as opposed to conceptual modeling primitives.

Some authors [4, 18] have identified three types of approaches to bridge the gap between an object model and a relational schema: (i) the "forward engineering" approach (also called object-relational mapping), in which the relational schema is generated from the class model that must be persisted (often together with the necessary code to propagate object persistence to the database); (ii) the "reverse engineering" approach (also called relational-object mapping), in which classes are produced from the existing relational structure; and (iii) the "meet-in-the-middle" approach, in which conceptual model and relational schema are designed, implemented and evolved separately, requiring some middleware to perform the correspondence between the objects and the database. Our approach is clearly positioned in the "forward engineering" camp.

6 Conclusions

The study of ontological foundations in conceptual modeling has produced a number of advances over the last decades. This paper extends some of these advances to relational schema design. We have shown that considering the ontological status of classes in a conceptual model makes it possible to conceive of novel transformation strategies that cannot be articulated with ontologically neutral conceptual modeling primitives. We have shown that the *one table per kind* strategy has performance characteristics that differ from the dominant approaches in the literature. Further, it supports multiple inheritance, orthogonal and overlapping hierarchies and dynamic classification.

We hypothesize that *one table per kind* can improve schema comprehension, as well as query writing and readability. This is because of the role that kinds play in cognition. There is a significant body of evidence in cognitive psychology [17, 26, 27], that object *kinds* are the most salient category of types in human cognition, being responsible for our most basic operations of object individuation and identity. Further, there is empirical evidence that the ontological distinctions employed here (those underlying OntoUML) contribute to improving the quality of conceptual models without requiring an additional effort to produce them [24]. In future work, we intend to assess whether the benefits trickle down to the system-level when coupled with the transformation strategy proposed here. We also intend to evaluate the impact of the various strategies on maintainability, in particular when considering the evolution of the relational schema (e.g., with the introduction/removal of classes, attributes, associations and the required data migration). Usability, maintainability and database performance analysis requires careful consideration of application-specific demands. Thus, considering application sensitivity is a clear issue for further work and application characteristics (e.g., demands on polymorphic queries) could guide the selection of a strategy.

Finally, recent developments have shown that there is a fruitful interplay between ontology-based techniques and database realization. For example, Ontology-Based Data Access (OBDA) approaches such as Ontop [5], have shown that is is possible to expose relational data in terms of a computational ontology. This is done in a meet-in-the-middle fashion by relying on the manual specification of a mapping from a computational ontology to an existing relational schema. We understand that synthesizing ODBA mappings with our approach is feasible, and would allow transparent ontology-based access to the produced relational schemas.

Acknowledgments

This research is partly funded by the Brazilian Research Funding Agencies CNPq (grants 312123/2017-5 and 407235/2017-5) and CAPES (23038.028816/2016-41).

References

1. Albano, A., Bergamini, R., Ghelli, G., Orsini, R.: An object data model with roles. In: Proc. 19th International Conf. on Very Large Data Bases. pp. 39–51. Morgan Kaufmann (1993)
2. Ambler, S.W.: Mapping objects to relational databases (1997), White Paper, AmbySoft Inc.

3. Ambler, S.W.: Agile database techniques: effective strategies for the agile software developer. Wiley (2003)
4. Cabibbo, L.: Objects meet relations: On the transparent management of persistent objects. In: Proc. CAiSE 2004. LNCS, vol. 3084, pp. 429–445. Springer (2004)
5. Calvanese, D. et al.: Ontop: Answering SPARQL queries over relational databases. Semantic Web **8**(3), 471–487 (2017)
6. Cardelli, L.: A semantics of multiple inheritance. In: Kahn, G., MacQueen, D.B., Plotkin, G.D. (eds.) Semantics of Data Types. LNCS, vol. 173, pp. 51–67. Springer (1984)
7. Carré, B., Geib, J.: The point of view notion for multiple inheritance. In: Yonezawa, A. (ed.) Proc. OOPSLA/ECOOP 1990. pp. 312–321. ACM (1990)
8. Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc. (2002)
9. Gottlob, G., Schrefl, M., Röck, B.: Extending object-oriented systems with roles. ACM Trans. Inf. Syst. **14**(3), 268–296 (1996)
10. Guarino, N., Guizzardi, G.: "We Need to Discuss the Relationship": Revisiting Relationships as Modeling Constructs. In: Proc. CAiSE. LNCS, vol. 9097, pp. 279–294. Springer (2015)
11. Guarino, N., Welty, C.A.: An overview of ontoclean. In: Handbook on Ontologies, pp. 201–220. Springer (2009)
12. Guizzardi, G.: Ontological foundations for structural conceptual models. Ph.D. thesis, University of Twente (10 2005)
13. Guizzardi, G., Figueiredo, G., Hedblom, M.M., Poels, G.: Ontology-based model abstraction. In: Proc. RCIS 2019. pp. 1–13. IEEE (2019)
14. Guizzardi, G., Fonseca, C.M., Benevides, A.B., Almeida, J.P.A., Porello, D., Sales, T.P.: Endurant types in ontology-driven conceptual modeling: Towards OntoUML 2.0. In: Proc. ER 2018. LNCS, vol. 11157, pp. 136–150. Springer (2018)
15. Ireland, C., Bowers, D., Newton, M., Waugh, K.: A classification of object-relational impedance mismatch. In: Proc.1st DBKDA. pp. 36–43 (March 2009)
16. Keller, W.: Mapping objects to tables: A pattern language. In: EuroPLoP 1997: Proc. 2nd European Conf. Pattern Languages of Programs. Siemens Tech. Report 120/SW1/FB (1997)
17. Macnamara, J.T., Macnamara, J., Reyes, G.E.: The logical foundations of cognition. No. 4 in Vancouver Studies in Cognitive Science, Oxford University Press on Demand (1994)
18. Philippi, S.: Model driven generation and testing of object-relational mappings. Journal of Systems and Software **77**, 193–207 (2005)
19. Rybola, Z.: Towards OntoUML for Software Engineering: Transformation of OntoUML into Relational Databases. PhD thesis, Czech Technical University in Prague (2017)
20. Rybola, Z., Pergl, R.: Towards OntoUML for software engineering: Transformation of kinds and subkinds into relational databases. Comput. Sci. Inf. Syst. **14**(3), 913–937 (2017)
21. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data Knowledge Engineering **35**(1), 83–106 (2000)
22. Steimann, F.: The role data model revisited. Applied Ontology **2**(2), 89–103 (2007)
23. Torres, A. et al.: Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. Information & Software Technology **82** (2017)
24. Verdonck, M., Gailly, F., Pergl, R., Guizzardi, G., Martins, B., Pastor, O.: Comparing traditional conceptual modeling with ontology-driven conceptual modeling: An empirical study. Information Systems **81**, 92 – 103 (2019)
25. Wieringa, R.J., de Jonge, W., Spruit, P.: Using dynamic classes and role classes to model object migration. TAPOS **1**(1), 61–83 (1995)
26. Xu, F.: From lot's wife to a pillar of salt: Evidence that physical object is a sortal concept. Mind & Language **12**(3-4), 365–392 (1997)
27. Xu, F., Carey, S.: Infants metaphysics: The case of numerical identity. Cognitive psychology **30**(2), 111–153 (1996)