frontiers | Frontiers in Computer Science

TYPE Original Research
PUBLISHED 08 December 2022
DOI 10.3389/fcomp.2022.1020168

Check for updates

OPEN ACCESS

EDITED BY
Heinrich C. Mayr,
University of Klagenfurt, Austria

REVIEWED BY
Evellin Cardoso,
Universidade Federal de Goiás, Brazil
Sara Migliorini,
University of Verona, Italy

*CORRESPONDENCE
João Paulo A. Almeida
jpalmeida@ieee.org

SPECIALTY SECTION
This article was submitted to Software,
a section of the journal
Frontiers in Computer Science

RECEIVED 15 August 2022
ACCEPTED 15 November 2022
PUBLISHED 08 December 2022

CITATION
Guidoni GL, Almeida JPA and Guizzardi G (2022) Preserving conceptual model semantics in the forward engineering of relational schemas.
Front. Comput. Sci. 4:1020168.
doi: 10.3389/fcomp.2022.1020168

Preserving conceptual model semantics in the forward engineering of relational schemas

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

[1]Ontology and Conceptual Modeling Research Group (NEMO), Federal University of Espírito Santo, Vitória, Brazil, [2]Federal Institute of Espírito Santo, Colatina Campus, Colatina, Brazil, [3]Services and Cybersecurity Group, University of Twente, Enschede, Netherlands

Forward engineering relational schemas based on conceptual models (in languages such as UML and ER) is an established practice, with several automated transformation approaches discussed in the literature and implemented in production tools. These transformations must bridge the gap between the primitives offered by conceptual modeling languages on the one hand and the relational model on the other. As a result, it is often the case that some of the semantics of the source conceptual model is lost in the transformation process. In this paper, we address this problem by forward engineering additional constraints along with the transformed schema (ultimately implemented as triggers). We formulate our approach in terms of the operations of "flattening" and "lifting" of classes to make our approach largely independent of the particular transformation strategy (one table per hierarchy, one table per class, one table per concrete class, one table per leaf class, etc.). An automated transformation tool is provided that traces the cumulative consequences of the operations as they are applied throughout the transformation process. We report on tests of this tool using models published in an open model repository.

KEYWORDS

conceptual model semantics, forward engineering, relational schemas, model-driven, semantics preservation

1. Introduction

Forward engineering relational schemas based on conceptual models such as (E)ER or UML class diagrams is an established practice, with a number of transformation strategies discussed in the literature (Torres et al., 2017; Guidoni et al., 2020) and commercial tools available and widely used in production settings. The approaches employed establish correspondences between the patterns in the source models and in the target relational schemas, as is typically the case in model-driven approaches. For example, in the *one table per class* approach (Keller, 1997; Fowler, 2002), a table is produced for each class in the source model; in the *one table per leaf class* approach (Philippi, 2005), a table is produced for each leaf class in the specialization hierarchy, with properties of superclasses accommodated at the tables corresponding to their leaf subclasses; in the *one table per hierarchy* approach (Torres et al., 2017), discriminator columns are introduced in the table corresponding to the superclass to identify which subclasses are instantiated by the entity represented in a row. An important benefit of all these approaches is the automation of an otherwise manual and error-prone schema design process. Automated transformations capture tried-and-tested design decisions, improving productivity, and the quality of the resulting schemas.

Despite the various benefits of automated transformation, there is often a mismatch between primitives of the conceptual modeling language and those of the target technical space; in this case, the relational model. Because of this, some of the information that was embodied in the conceptual model may be lost in the transformation process. In the case of the transformation of specialization hierarchies, depending on the class-to-table transformation strategy, invariants that are expressed directly in the conceptual model through cardinality constraints and association end typing may be weakened. For example, consider a hierarchy with a class `Person` at the top and a single subclass `Student` with a mandatory attribute `enrollmentNumber`. In the *one table per hierarchy* approach, mandatory attributes of subclasses (in this case `enrollmentNumber`) are implemented as *nullable* columns of the table representing the whole hierarchy (in this case of the table representing all persons). This is required in this approach, because not all persons have enrollment numbers—albeit all students do. This kind of cardinality constraint present in the source model is no longer enforced by the target relational schema, which admits a non-student person to be assigned an `enrollmentNumber` and a student to have a null `enrollmentNumber`. Foreign keys in other tables that should identify only students (for example concerning the borrower of a book in the university library) now identify persons indistinctly.

The first contribution of this paper is to identify the constraints that are lost in the transformation of structural conceptual models to relational schemas, independently of the class-to-table transformation strategy applied. Our approach is based on the primitive refactoring operations of *flattening* and *lifting* of classes that can account for the transformation of a specialization hierarchy into tables according to various class-to-table transformation strategies in the literature (Guidoni et al., 2020). Secondly, we show how to incorporate the generation of the required constraints along a transformation of a conceptual model into a relational schema. As the transformation advances, at each application of an operation, we maintain a set of traces from source to target model, ultimately producing not only the relational schema, but also invariants that are applicable to the resulting schema using the set of traces. These invariants are implemented as triggers, detecting when data that violates the conceptual model is introduced in the database. A fully automated implementation of the transformation is provided.

This paper is further structured as follows: Section 2 discusses extant approaches to the transformation of a structural conceptual model into a target relational schema, presenting the primitive operations of flattening and lifting that can account for the various transformation strategies (Guidoni et al., 2020); it further provides an example conceptual model transformed with two strategies for illustration. Section 3 discusses flattening and lifting in further detail, presenting the aspects of the semantics that are lost in the application of each of these operations, identifying thus the constraints that should be added in order to preserve the source model semantics. (Formal specifications of the operations subjected to automated proof are provided in Supplementary material). Section 4 shows how we instrument the transformation process by maintaining a set of traces that are updated in each flattening and lifting application; these traces are ultimately used to generate triggers in the database that reflect the constraints that are required by each step in the transformation process; the automatically-generated triggers accumulate all the required constraints. Section 5 reports on the implementation of the tool and on the tests that were performed to validate the approach, using models made available in an open model repository (Barcelos et al., 2022). Section 6 discusses related work, and, finally, Section 7 presents concluding remarks.

2. Background

2.1. Extant approaches

The relational model does not directly support constructs corresponding to class generalization and specialization, and, hence, realization strategies are required to cope with the use of these notions in source conceptual models. Such strategies are described by several authors (Keller, 1997; Fowler, 2002; Ambler, 2003; Shah and Slaughter, 2003; Philippi, 2005; Torres et al., 2017; Guidoni et al., 2020) under various names. We include here some of these strategies for the purpose of exemplification.

One common approach is the *one table per class* strategy, in which each class gives rise to a separate table, with columns corresponding to the class's features. Specialization between classes in the conceptual model is emulated and gives rise to a foreign key in the table that corresponds to the subclass. This foreign key references the primary key of the table corresponding to the superclass. This strategy is also called "class-table" (Fowler, 2002), "vertical inheritance" (Torres et al., 2017), or "one class one table" (Keller, 1997), and is also very common in the EER transformation literature (Teorey et al., 1986; Markowitz and Shoshani, 1992). 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.

A common variant of this approach is the *one table per concrete class* strategy (Philippi, 2005). In this case, an operation of "flattening" is applied for the abstract superclasses. In a nutshell, "flattening" removes a class from the hierarchy by transferring its attributes and relations to its subclasses. This reduces the number of tables required to read or to insert all attributes of an instance, but introduces the need for unions in polymorphic queries involving the flattened abstract classes.

The extreme application of "flattening" to remove all non-leaf classes of a taxonomy yields a strategy called *one table per leaf class*. In this strategy, also termed "horizontal inheritance" (Torres et al., 2017), 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.

A radically different approach is the *one table per hierarchy* strategy, also called "single-table" (Fowler, 2002) or "one inheritance tree one table" (Keller, 1997). It can be understood as the opposite of *one table per leaf class*, applying a "lifting" operation to subclasses instead of the "flattening" of superclasses. Attributes of each subclass become optional columns in the superclass table. 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 our previous work (Guidoni et al., 2020), we have proposed a novel approach based on the combined and selective use of both "lifting" and "flattening." This approach leverages the specialized semantics of the OntoUML (Guizzardi, 2005) conceptual modeling language to precisely determine which classes should be flattened and which should be lifted—given a particular set of conceptualization choices made about the domain. This is possible because OntoUML explicitly represents in a system of stereotypes a number of finer-grained ontological distinctions among types of classes. These stereotypes enable the explicit representation of these choices.

Among these distinctions, a special attention is given to *kinds*, i.e., special sorts of static classes that represent the essential types of things in that domain. They tessellate the space of existing entities at any given point in time, i.e., the kinds in an OntoUML model are both mutually disjoint and together exhaust the set of instances of that model. In other words, every instance of an OntoUML model belongs to exactly one kind and always to that same kind. Thus, kinds constitute a static backbone of the conceptual model, which is then both refined into subclasses and abstracted into superclasses. On one hand, they are typically specialized by dynamic classes characterized by contingent properties that their instances can bear in some situations but not in others (e.g., a person can contingently be a teenager or a student). When these dynamic classes are characterized by intrinsic properties, they are called *phases* (e.g., teenager, living person); when by relational properties, they are called *roles* (e.g., student, father, husband). On the other hand, multiple kinds are refactored into *abstract classes* that represent properties that are shared by their instances. These abstract classes can also be either static (called *categories*) or dynamic (called *role mixins*). Since every instance of a domain must instantiate a kind, the use of these refactoring classes in a model can lead to multiple classification schemes (e.g., an entity can be both of the kind car but also instantiate the category physical object). In other words, entities can instantiate several classes but one unique kind and, hence, classes can specialize multiple superclasses but at most one kind.

The approach discussed in Guidoni et al. (2020) leverages these distinctions to automatically and in a reiterated way: (a) "flatten" all the properties in the abstract classes until they are projected into the specializing concrete classes; (b) in sequence, "lift" all the properties of these concrete classes up to their generalizing kinds. As result, the model concentrates all the original information around kinds, which are then mapped into relational tables. We call this strategy *one table per kind*.

Finally, we highlight that the operations of "flattening" and "lifting" are the same in all these different transformation strategies. What varies across strategies are the different classes that these operations target as focal points, and the different orders in which they are iterated. All these strategies can be conceived as complex procedures combining the very same primitive operations of "flattening" and "lifting."

2.2. Example transformations

In order to illustrate the consequences of the transformation strategy selected on the resulting relational schema, we apply here two different transformations to the (Onto)UML class diagram shown in Figure 1. In this model, the abstract class `NamedEntity` is specialized into concrete classes `Person` and `Organization`. The abstract class `Customer` is specialized into concrete classes `PersonalCustomer` and `CorporateCustomer`. `Persons` are classified dynamically according to life phase, and must be either in the `Child` or the `Adult` phase. In the `Adult` phase, a `Person` can enter into a `SupplyContract` (a *relator* kind) playing the role of `PersonalCustomer` and thus hiring an `Organization` as a `Contractor`. `Organizations` can also assume the role of `CorporateCustomer` in a `SupplyContract`. The transformation approaches selected for illustration are *one table per concrete class* and *one table per kind*, and are used throughout the paper. As discussed in the previous sub-section, the former relies on the flattening of abstract classes (as they appear in standard UML models). The latter makes use of OntoUML stereotypes (Guizzardi et al., 2018) to determine when to apply flattening and lifting.

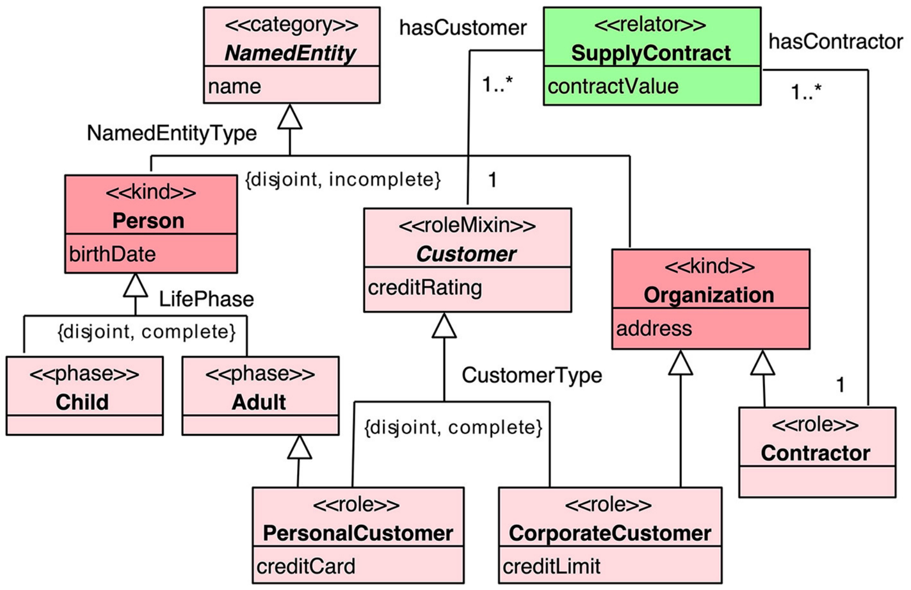


FIGURE 1
Example source conceptual model.

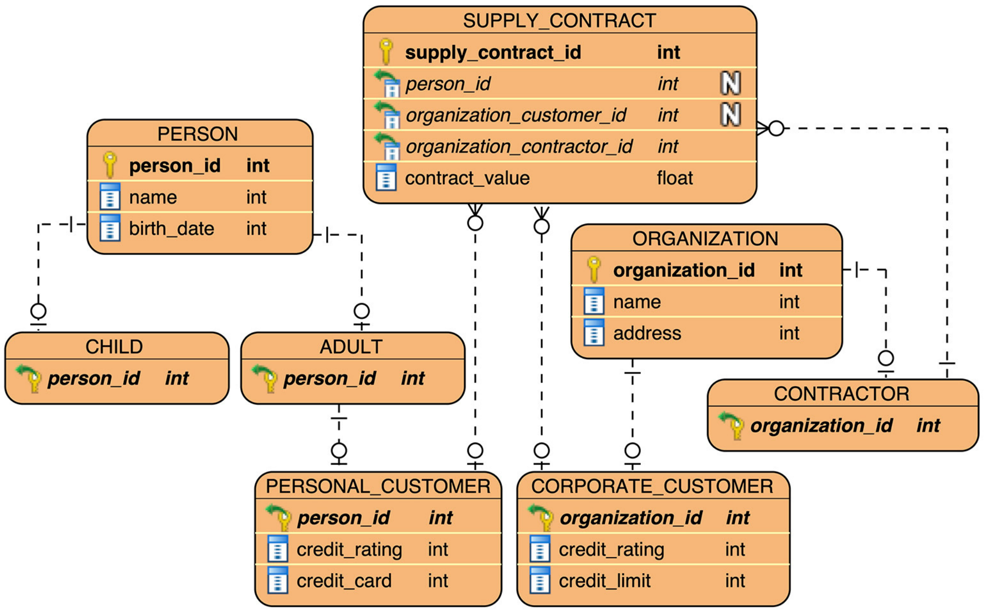


FIGURE 2
Target schema with the *one table per concrete class* strategy.

Figure 2 presents the resulting schema in the *one table per concrete class* strategy. The abstract classes `NamedEntity` and `Customer` have been flattened out, and a table is included for each concrete class. Foreign keys are used to emulate inheritance (e.g., the table `ADULT` corresponding to the concrete class `Adult` has as its primary key a foreign key `person_id` which identifies an entry in the table `PERSON`). Joins are required to access all attributes of an instance of a subclass in tandem (e.g., to obtain the `name` and `credit_rating` of an instance of `PersonalCustomer` represented in the database).

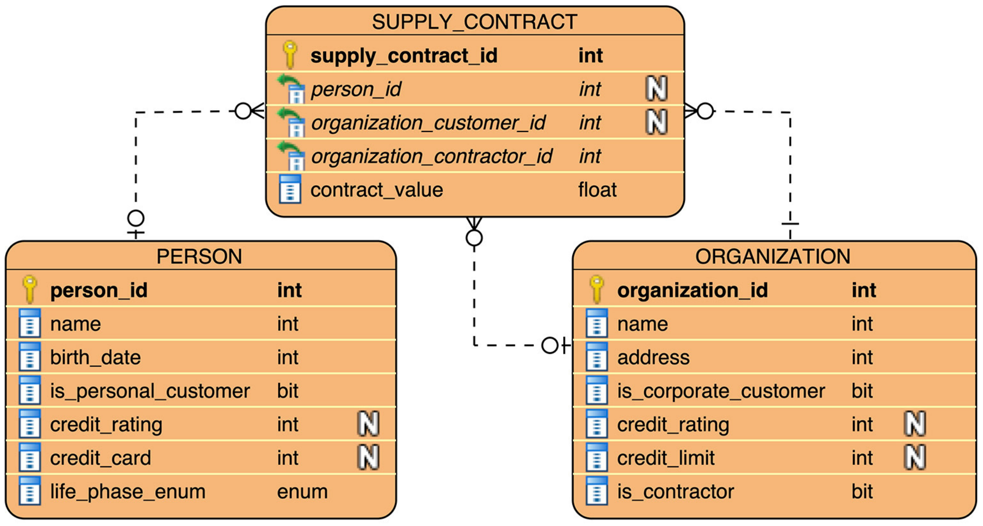


FIGURE 3
Target schema with the *one table per kind* strategy.

Figure 3 presents the result of applying the *one table per kind* strategy. Again, `NamedEntity` and `Customer` (abstract classes) are flattened out. In addition, all concrete classes are lifted until only one concrete class (the kind) remains. There is no emulation of inheritance with foreign keys; discriminators are used instead to identify the subclasses that are instantiated by an entity. No joins are required to access jointly attributes of an entity. A number of nullable columns are introduced as the result of "lifting" (namely `credit_rating`, `credit_card`, `credit_limit` due to the migration of attributes in lifting; and `person_id` and `organization_customer_id` due to the migration of association ends opposite of `SupplyContract` in flattening and lifting).

3. Bridging the semantic gap

We have observed during our investigations into the various transformation strategies in the literature that they could all be understood as complex procedures that could be formulated in terms of lifting and flattening. Here, we present these operations in detail, including their consequences to the existing attributes and associations in the model.

Using the terminology of Lúcio et al. (2016) and Lano et al. (2018), both operations can be understood conceptually as *endogenous* model *refactorings*. An endogenous transformation is one whose target and source models are represented in the same language (here they are both class diagrams); a refactoring is a transformation that performs *update-in-place* making local changes to a part of the model. These are usually motivated with the *model editing* intent (Lúcio et al., 2016), although here, they have been motivated by progressing toward a model that can be more easily *translated* in an ultimate exogenous model transformation (one that produces a relational schema from a class diagram). Because of that, in some cases, our operations have the opposite intent of some refactoring operations in the literature. For example, flattening is the opposite of "extract class" of Fowler et al. (2012). The operations are similar to those defined by Guizzardi et al. (2019), but are used there solely for the purpose of model abstraction (and hence, the authors of that work were not concerned with the loss of constraints in the application of their operations). ["Flattening" as used here should not be mistaken with the homonymous model transformation *pattern* of Iacob et al. (2008), which refers to the elimination of hierarchical containment structures in the abstract syntax.] The operations are applied automatically once the classes to be flattened or lifted in a step are identified.

We assume here that the source conceptual model consists of (abstract and concrete) classes, attributes with multiplicity constraints, binary associations also with multiplicity constraints at both association ends, generalizations between classes, and generalization sets with varying `isCovering` and `isDisjoint` metaproperties following the UML metamodel[1]. Multiplicities are represented for simplicity with lower bound cardinalities corresponding to either `0` or `1`, and upper bound cardinalities corresponding to `1` or `*`. The approach we propose is not specific to OntoUML. In fact, the proposed operations of lifting and flattening are general to all kinds of structural conceptual models.

1 https://www.omg.org/spec/UML/.

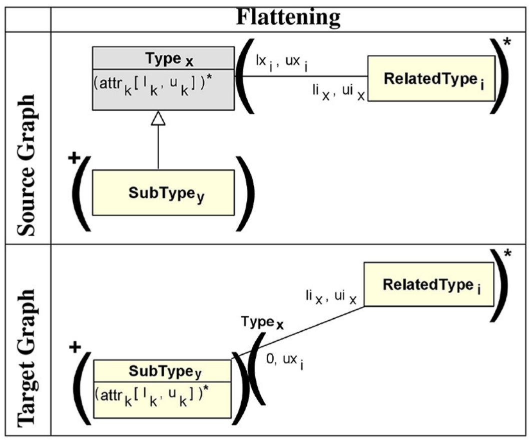
3.1. Flattening

In the flattening operation, shown schematically in Table 1, every attribute of a class that is to be removed from the model ($Type_x$ in gray) is migrated to each of its direct subclasses ($SubType_y$). Association ends attached to the flattened superclass are also migrated to the subclasses (creating new associations in the process, one for each subclass). The lower bound cardinality of the migrated association end (lx_i) is relaxed to `0`, as the original lower bound is not necessarily satisfied for each of the subclasses. The cardinalities of attributes ($attr_k$) as well as association ends attached to classes *other than* the flattened class ($RelatedType_i$) are maintained, as these are invariants that apply to all subclasses in virtue of the semantics of specialization. (For simplicity, the diagram represents only one association between $Type_x$ and a $RelatedType_i$, but in fact, there can be many such associations).

Preconditions to the application of flattening are that the class is at the top of the present hierarchy (i.e., it has no superclass) and that is either abstract or all its subclasses are complete specializations of it. (In the particular case of a single subclass of an abstract class, flattening is trivialized; in any case, the situation indicates a clear model smell).

Figure 4 shows the result of the application of the flattening operation on the abstract classes in the example in Figure 1, with changes highlighted in red boxes. The `Person` and `Organization` classes now have a `name` attribute, due to the flattening of `NamedEntity`. As a consequence of the flattening of `Customer`, `PersonalCustomer`, and the `CorporateCustomer` now have an attribute `creditRating`. Further, the `SupplyContract` class is now related directly to the `PersonalCustomer` and the `CorporateCustomer` classes. A minimum cardinality of `0` is used in the association ends connected to `PersonalCustomer` and `CorporateCustomer`.

TABLE 1 The flattening operation.



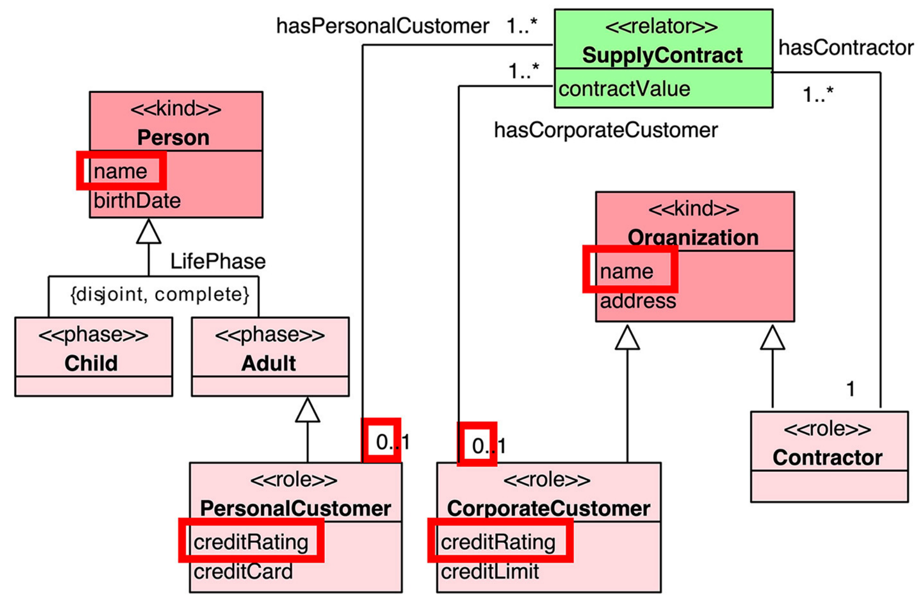


FIGURE 4
Result of flattening `NamedEntity` and `Customer`.

3.1.1. Missing constraints due to association end migration in flattening and relaxed lower bounds

The following constraints are missing from the resulting model after the operation: (**MC1**), in the presence of an association in the source model with an association end attached to the flattened superclass and lower bound $lx_i = 1$, we must ensure that at least one instance of the target subclasses is related to an instance of the *RelatedType*$_i$ through the associations that were introduced in the flattening operation, in order to enforce the original lower bound cardinality. In this case, for a particular `SupplyContract`, there is a related `PersonalCustomer` or a `CorporateCustomer`; and (**MC2**), in the presence of an association in the source model with an association end attached to the flattened superclass and upper bound $ux_i = 1$, we must ensure that an instance of the superclass is related through the associations that were introduced in the flattening operation *to at most one instance of the union of all subclasses*. In this case, considering both conditions, an instance of `SupplyContract` is related to *exactly one instance in the union of* `PersonalCustomer` and `CorporateCustomer`. A proof that these required (missing) constraints **MC1** and **MC2** indeed follow from the semantics of the original model and the definition of flattening is available in the Supplementary material. We employed automated theorem provers in the TPTP system. See `flattening.p`, in the scope of the flattening of `Customer` for the formalization in first-order logics in the TPTP syntax (and corresponding formalization in CLIF). Proof reports are available (Vampire 4.6.1 Automated Theorem Prover).

3.2. Lifting

In the lifting operation, shown in Table 2, every attribute of the class that is lifted (*SubType*$_y$ in gray) is migrated to each direct superclass, with lower bound cardinality (l_k) relaxed to `0` (i.e., mandatory attributes become optional). Upper bound cardinality (u_k) is maintained. Association ends attached to the lifted class are migrated to each direct superclass. The lower bound cardinality constraints of the association ends attached to classes *other than* the lifted class (*RelatedType*$_i$) (if any) are relaxed to `0` in the same vein of the attributes of the lifted subclass. When no generalization set is present a Boolean attribute is added to each superclass, to indicate whether the instance of the superclass instantiates the lifted class (*isSubType*$_y$).

If a generalization set is used, a discriminator enumeration is created (GS_a) with labels corresponding to each *SubType*$_y$ of the generalization set (see Table 3). An attribute with that discriminator type is added to each superclass (gs_a). Its cardinality follows the generalization set: it is optional for incomplete generalization sets (and mandatory otherwise); and multivalued for overlapping generalization sets (and monovalued otherwise).

A precondition to the application of lifting is that the class is a leaf of the present hierarchy (i.e., it has no subclass), and, if it is a subclass in a generalization set, its siblings in the generalization set must also be leaves of the present hierarchy. Approaches that rely on lifting (such as *one table per hierarchy*) usually rule out multiple inheritance, since in the presence of multiple inheritance, the preservation of the cardinalities (ly_i and uy_i becomes problematic) in the lifting step. Here, we operate under the assumption that multiple inheritance is admissible, but that further lifting steps will end up consolidating the various associations introduced due to lifting to various subclasses into a single one; in other words, we assume that lifted classes are ultimately indirect specializations of a single class. In the *one*

TABLE 2 The lifting operation.

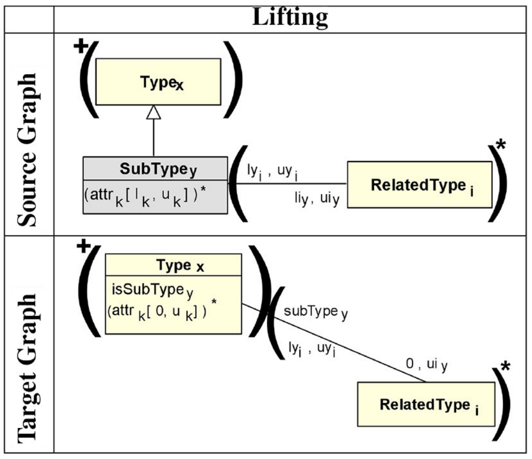
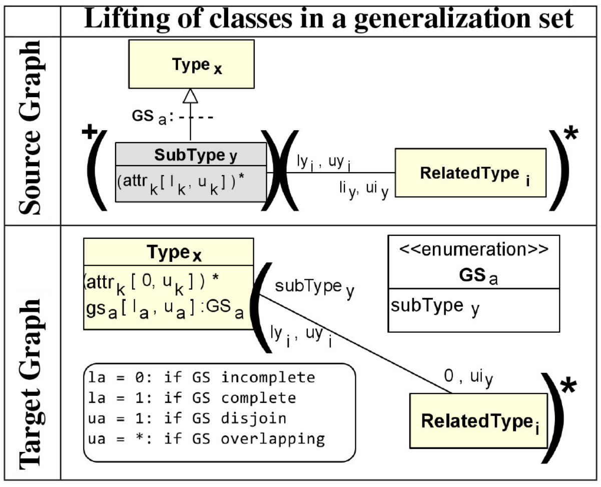


TABLE 3 The lifting operation with a generalization set.



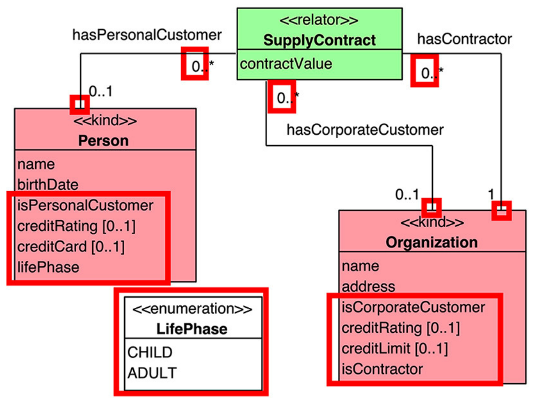


FIGURE 5
Result of the lifting of `Child`, `Adult`, `PersonalCustomer`, `CorporateCustomer`, and `Contractor`.

table per hierarchy approach, this means that hierarchies are disjoint (i.e., there is no class that specializes more than one top-level class); in the *one table per kind* approach, this means that kinds are disjoint (i.e., there is no class that specializes more than a class).

The result of the lifting of `CorporateCustomer`, `PersonalCustomer`, `Child`, `Adult`, and `Contractor` is shown in Figure 5. The lifting of `Contractor` to `Organization` adds a discriminator `isContractor` to that superclass and leads to a relaxed minimum cardinality of `0` in the association end of `hasContractor` attached to `SupplyContract`. This is because not all organizations are involved in a `SupplyContract`. The lifting of `CorporateCustomer` adds a discriminator `isCorporateCustomer` to `Organization` along with the (now optional) attributes `creditRating` and `creditLimit`. The figure shows the accumulated results of lifting of `PersonalCustomer` to `Adult` and the lifting of `Adult` to `Person`. The lifting of the `LifePhase` generalization set leads to the introduction of a `lifePhase` attribute typed with a corresponding enumeration. The multiplicity of the discriminator `lifePhase` is `1` given the original generalization set is `{disjoint, complete}` (we assume the cardinality `[1..1]` is the default multiplicity, and hence omit it from mandatory monovalued attributes in all diagrams).

3.2.1. Missing constraints due to lifted attributes

The lifting of attributes has the following consequences: (**MC3**) mandatory attributes are now optional, a situation which, although necessary—since not all instances of the superclass are instances of the lifted subclass—is inadmissible for instances of the lifted subclass according to the original model. The example model after lifting in Figure 5 admits the possibility that a personal customer represented as an instance of `Person` is assigned no value for `creditRating`. Further, (**MC4**) instances of the superclass (even those that do not represent instances of a lifted subclass) may indiscriminately be given values to the lifted attributes. In the example model, it is possible for a child represented as an instance of `Person` to be assigned a value for `creditCard`, even though this was inadmissible in the original model. The creation of a discriminator attribute in the lifting process provides us the means to formulate missing constraints to enforce the semantics of the original model: assignment of a value to a lifted attribute must be admitted conditionally on the value of the discriminator. In the model shown in Figure 5, due to the lifting of `CorporateCustomer` to `Organization`, we need to condition the assignment of `creditRating` and `creditLimit` to those instances of `Organization` for which `isCorporateCustomer=true`. Since we have two applications of lifting of `PersonalCustomer` to `Adult` and then from `Adult` to `Person`, we have the condition that persons may be assigned a `creditRating` and a `creditCard` only when `isPersonalCustomer=true` (i) and that is only admissible when `lifePhase=ADULT` (ii). Note that discriminator attributes may also be subject to lifting in further applications of the operation, and will also result in additional occurrences of MC4 in those cases.

3.2.2. Missing constraints due to migration of association ends to superclasses

The lifting operation results in the migration of association ends from the lifted subclasses to their superclasses. As a consequence, analogously to mandatory attributes, the association ends opposite to the superclass must have their lower bound cardinality (li_y) relaxed. Again, the discriminator attribute ($isSubType_y$ or gs_a) gives us the means to enforce the semantics of the association is in the source model: (**MC5**), in case $li_y = 1$, we need to introduce a constraint that, for each instance of the superclass, if $isSubType_y$ = `true` (or gs_a is equal to or includes $subType_y$), then that instance of the superclass must be associated to one instance of the related class ($RelatedType_i$). (**MC6**), only those instances with $isSubType_y$ = `true` (or gs_a is equal to or includes $subType_y$) may be associated to an instance of the related class ($RelatedType_i$), to ensure the association end typing is respected. In the model shown in Figure 5, due to the lifting of `CorporateCustomer` to `Organization`, we need to condition the association of an organization to a supply contract through `hasCorporateCustomer` to those instances of `Organization` for which `isCorporateCustomer=true`. Likewise, due to the lifting of `Contractor` to `Organization`, we need to condition the association of an organization to a supply contract through `hasContractor` to those instances of `Organization` for which `isContractor=true`. Since we have two applications of lifting of `PersonalCustomer` to `Adult` and then from `Adult` to `Person`, we have the condition that persons must be associated with one `SupplyContract` when `isPersonalCustomer=true` (i) and that is only admissible when `lifePhase=ADULT` (ii). A demonstration that these required (missing) constraints (**MC3–MC6**) indeed follow from the semantics of the original model and the definition of lifting is available in the Supplementary material (see `lifting.p`, in the scope of the lifting of `CorporateCustomer`).

3.3. Translating generalizations into schemas

The literature on EER to relational schema mapping—such as the seminal work of Teorey et al. (1986)—has early on identified missing integrity constraints when translating EER diagrams with generalization and subset hierarchies into relational schemas. They have concentrated their efforts in the *one table per entity* strategy, which does not apply any operations such as flattening and lifting. In any case, the missing constraints they have identified must be taken into account in any transformation that still preserves generalizations in the last translation step (e.g., *one table per class*, *one table per concrete class*).

In these transformations, there is an emulation of generalization in the relational schema. A table corresponding to a subclass has as primary key a foreign key that refers to entries in the table corresponding to the topmost superclass. (In the case of multiple inheritance of classes in the top of the hierarchy, this becomes a composite key). Let us consider for instance the model in Figure 4, which is the result of the flattening of abstract classes, and hence is the class model to be translated into a relational schema in the *one table per concrete class approach*. In the translation of this model (depicted earlier in Figure 2), the table `ADULT` corresponding to the concrete class `Adult` has a foreign key `person_id` which is the primary key of `PERSON`, and likewise for `CUSTOMER` and `PERSONAL_CUSTOMER`. A similar solution applies to the other subclasses in the model.

3.3.1. MC7 missing constraint due to emulation of a generalization

The emulation of generalization provides some guarantees of referential integrity through the use of keys: an entry representing a subclass instance will always be properly related to its superclasses. However, the following missing integrity constraint is identified by Teorey et al. (1986) (Section 3.1.4, with terminology adjusted here): when a generalization set is disjoint, it must be inadmissible for two tables corresponding to disjoint subclasses to have entries that refer to the same "superclass key." This is required to preserve the semantics of disjointness and applies in our example, to the tables `CHILD` and `ADULT` in the *one table per concrete class* approach. Note that this constraint is not required in approaches that remove all generalizations by progressively applying flattening and lifting (e.g., *one table per leaf class*, *one table per hierarchy*, and *one table per kind*).

4. The instrumented transformation

As we have observed in the previous section, the operations produce cumulative effects throughout the transformation process. We discuss here how this process is operationalized in our implementation by maintaining a set of traces, which are updated when each of the two operations are applied. We also show the last step of the overall conceptual model to relational schema transformation, which entails the production of triggers to enforce the accumulated constraints from the final set of traces and the refactored model.

4.1. Transformation process

Three class-to-table approaches have been implemented in a plugin to Visual Paradigm[2]: *one table per class*, *one table per concrete class*, and *one table per kind*. There are no manual steps in our approach, and the implementation applies the three transformation strategies fully automatically. The three approaches are accommodated by applying flattening and lifting operations in different orders and under different conditions. In the *one table per kind* and in *one table per concrete class*, first, all abstract classes are flattened, from the top of the hierarchy until concrete classes are reached. In the *one table per kind*, leaf concrete classes are lifted until all only kinds remain. In all strategies, the resulting refactored class model is then translated into a relational schema. (In the case of the *one table per class* approach, no operations of flattening and lifting are performed, and hence translation happens in the original model).

4.2. Traces

We have employed a trace table following the model shown in Figure 6. A `TraceTable` is produced in each application of the transformation and consists of a collection of `TraceSets` for each source class in the model. Each `TraceSet` identifies the set of `Traces` to target classes. The transformation begins by initializing the `TraceTable` with one `TraceSet` for each

2 See https://github.com/nemo-ufes/forward-db-vp-plugin for the full implementation.

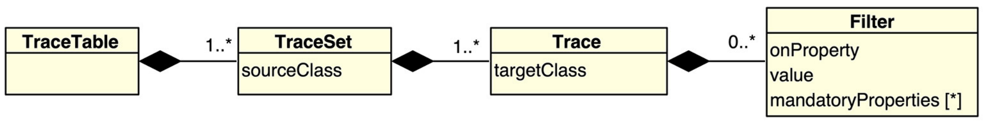


FIGURE 6
Trace table structure.

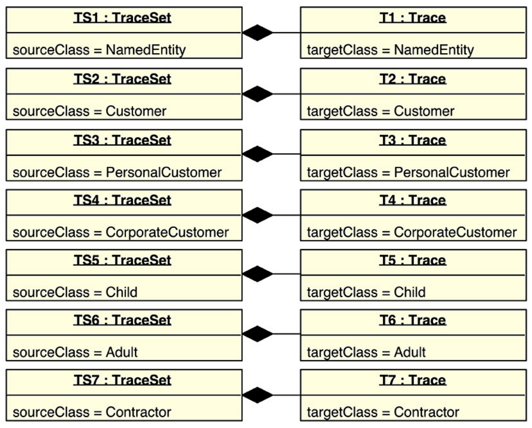


FIGURE 7
Initial trace table.

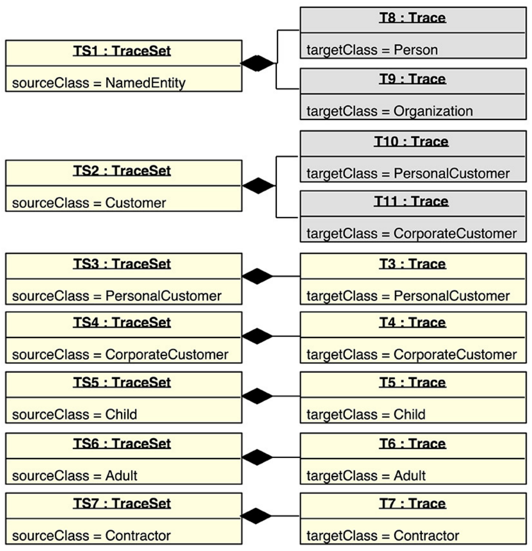


FIGURE 8
Trace table after the flattening of `NamedEntity` and `Customer`.

source class in the model each of which containing a single `Trace` to the same class. As the transformation progresses, traces in a trace set are updated. Required `Filter`s are added in case of lifting, as will be detailed in the sequel.

Figure 7 shows the trace sets contained in the initial `TraceTable` as an object diagram. For brevity, the traces for `Person`, `Organization`, and `SupplyContract` classes will be omitted as they remain unchanged throughout the process to apply the *one table per kind* approach, which we adopt in this example, as it applies both flattening (with an intermediate result that reflects *one table per concrete class*) and lifting.

In our running example, `NamedEntity` will be flattened to the `Person` and `Organization` classes. This means that `NamedEntity` will be identified by the union of all instances of `Person` and `Organization` (rule presented in Section 3.1). Thereby, is necessary to replace the traces that currently refer to the flattened class with those referring to the subclasses in the flattening operation. The same is true for the `Customer` class and its subclasses. The result of this updates are shown in gray in Figure 8. (If we are following the *one table per concrete class* approach, this would be the final state of the trace table).

When lifting is performed on a subclass (to carry out *one table per kind* strategy at this point in the transformation), the traces currently referring to the lifted subclass are updated with traces to the superclasses. Filters are added according to the discriminators required as discussed in Section 3. The set of mandatory properties that were lifted are added to the filter. For example, the lifting of the`PersonalCustomer` class indicates that it becomes identified in the `Adult` class when the `isPersonalCustomer` attribute is equal to `true`, requiring the filling of the `creditRating` and `creditCard` attributes. As the `PersonalCustomer` class no longer exists in the resulting model, every reference to it in the trace table is updated to the target class of the lifting process (in this case, `Adult`). Thereby, the `PersonalCustomer` and `Customer` source classes that traced `PersonalCustomer` now trace `Adult` (including filters). Figure 9 shows the result of the lifting of `PersonalCustomer`, `CorporateCustomer`, and `Contractor`, whose lifting preconditions are established. The trace for source class `CorporateCustomer` is updated to `Organization`. The traces of source class `Customer` is also updated, as it targeted the lifted classes. It now includes a trace to `Adult` and a trace to `Organization`, since the previously traced `PersonalCustomer` and `CorporateCustomer` have been lifted to those classes. The trace for `Contractor`

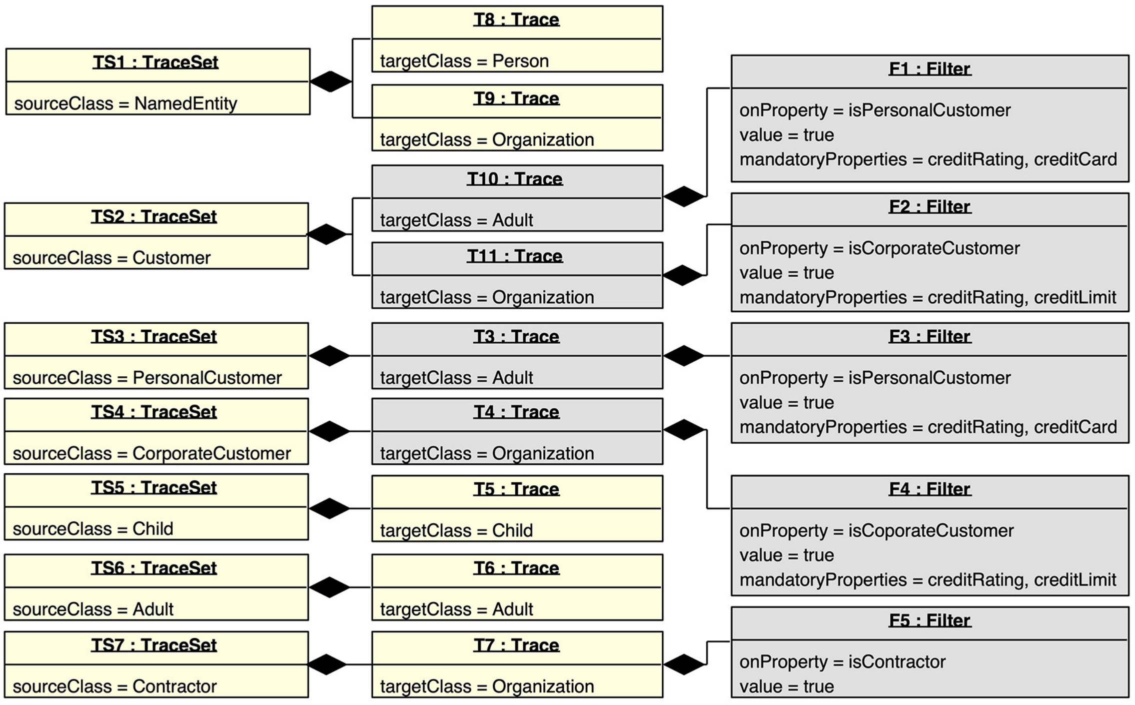

TS1 : TraceSet
sourceClass = NamedEntity
T8 : Trace
targetClass = Person
T9 : Trace
targetClass = Organization
TS2 : TraceSet
sourceClass = Customer
T10 : Trace
targetClass = Adult
T11 : Trace
targetClass = Organization
TS3 : TraceSet
sourceClass = PersonalCustomer
T3 : Trace
targetClass = Adult
TS4 : TraceSet
sourceClass = CorporateCustomer
T4 : Trace
targetClass = Organization
TS5 : TraceSet
sourceClass = Child
T5 : Trace
targetClass = Child
TS6 : TraceSet
sourceClass = Adult
T6 : Trace
targetClass = Adult
TS7 : TraceSet
sourceClass = Contractor
T7 : Trace
targetClass = Organization
F1 : Filter
onProperty = isPersonalCustomer
value = true
mandatoryProperties = creditRating, creditCard
F2 : Filter
onProperty = isCorporateCustomer
value = true
mandatoryProperties = creditRating, creditLimit
F3 : Filter
onProperty = isPersonalCustomer
value = true
mandatoryProperties = creditRating, creditCard
F4 : Filter
onProperty = isCoporateCustomer
value = true
mandatoryProperties = creditRating, creditLimit
F5 : Filter
onProperty = isContractor
value = true

FIGURE 9
Trace table after the lifting of `PersonalCustomer`, `CorporateCustomer` and `Contractor`.

is updated to `Organization`. In all cases filters are added to identify the subclass in the superclass.

Finally, Figure 10 shows the lifting of `Child` and `Adult`. Note that since `Adult` already had a filter (in the trace set of `PersonalCustomer`), this filter is preserved, and the new one is added (`lifePhase=ADULT`). This is because the final conditions that must be imposed on the target class are the conjunction of these filters. At this point, no further operations are applicable in the *one table per kind* approach and we have the final state of the trace table for translation to the relational schema. Each class in the set of target classes referred to in the trace table corresponds to a table in the resulting relational schema.

4.3. Generation of triggers

We focus here on the generation of triggers to detect violations of the missing constraints that were identified in Section 3. The trigger code is generated by using the final state of the trace table and the source model.

4.3.1. Consequences of lifting

The process goes through the trace table to identify the target classes that have filters that were the result of the lifting of mandatory attributes; for example, `isPersonalCustomer` for the target class `Person` and `isCorporateCustomer` for `Organization`. A trigger specification must then detect violations: if (i) the discriminator attribute in the filter matches the filter value and at the same time any of the columns corresponding to mandatory attributes are not filled in (line 6 in Listing 1 for the `PERSON` table and the filter on `isPersonCustomer`), addressing the missing constraint **MC3**; or, if (ii) the discriminator attribute in the filter does not match the filter value and any of the columns corresponding to mandatory attributes are filled in (line 8 in Listing 1), addressing the missing constraint **MC4**. (While the trigger shown in the listing applies to inserts on `PERSON`, a trigger with the same body is included for any subsequent updates. The listing of that trigger is omitted here for brevity).

```
1  delimiter //
2  CREATE TRIGGER tg_person_i BEFORE INSERT ON person
3  FOR EACH ROW
4  BEGIN
5      declare msg varchar(128);
6     if (
7          (NEW.is_personal_customer = true AND (NEW.
      credit_card is null OR NEW.credit_rating is null))
8          OR
9          (NEW.is_personal_customer <> true AND (NEW.
      credit_card is not null OR NEW.credit_card is not
      null))
10     )
11     then
```

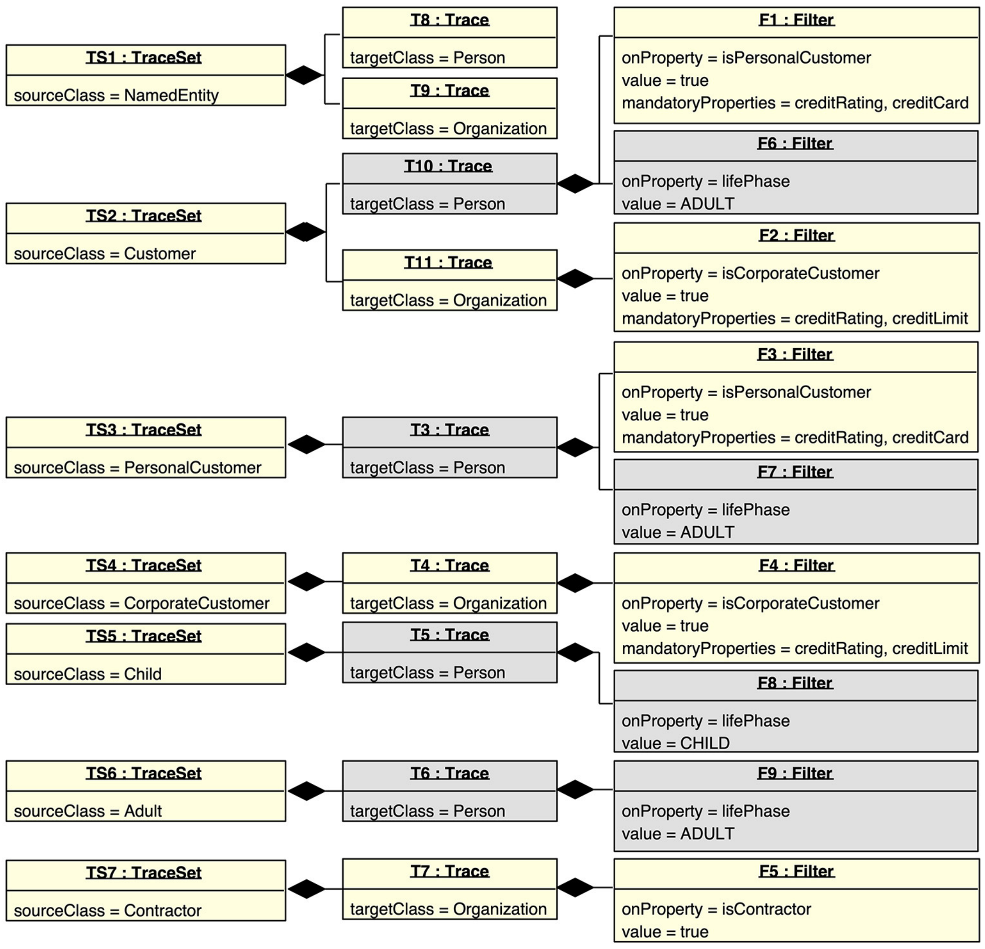


FIGURE 10
Trace table after the lifting operation on the generalization set with `Child` and `Adult`.

```
        set msg = 'ERROR: ...[tg_person_i].';
        signal sqlstate '45000' set message_text = msg
     ;
    end if;

    if(
        ( NEW.life_phase_enum <> 'ADULT' AND (
                      (NEW.is_personal_customer is not
     null  AND  NEW.is_personal_customer = TRUE ) OR
                        NEW.credit_rating is not null  OR
     NEW.credit_card is not null   ) )
    )
    then
        set msg = 'ERROR: ...[tg_person_i].';
        signal sqlstate '45000' set message_text = msg
     ;
    end if;
END; //
delimiter ;
```
Listing 1 Trigger for the PERSON table.

The process also identifies, for each target class (here `SupplyContract`, `Person`, and `Organization`), whether they were originally associated with other classes in the source model. When the associated source classes have filters in their respective trace sets (here `Contractor` and `Customer` for `SupplyContract`), this means the associated source classes were subject to lifting, and possible violations of the original semantics need to be detected according to the missing constraints identified in Section 3.2.2. The following violations must be detected in the trigger: (i) if an entry in the table corresponding to the target class (such as `SUPPLY_CONTRACT`) is associated to a lifted source class (such as `Contractor`), but the required filters associated to that source class are not satisfied (lines 7–19 of Listing 2 for `SUPPLY_CONTRACT` in its original association with a `Contractor`; lines 21–33 for the original association with a `Customer` concerning its trace to `Organization`, and lines 35–48 for the original association with a `Customer` concerning its trace to `Person`). This addresses **MC6**.

4.3.2. Consequences of flattening

The process also goes through the trace table to identify those trace sets of originally associated classes with more than one trace, which is a consequence of flattening. In our example, this occurs for the association between `SupplyContract` and `Customer`. As discussed in Section 3.1 (under "Missing constraints"), we must ensure that a supply contract is associated with exactly one instance in the union of `PersonalCustomer` and `CorporateCustomer` in lines 50–56 of Listing 2. This addresses the missing constraints **MC1** and **MC2**.

```
1  delimiter //
2  CREATE TRIGGER tg_supply_contract_i BEFORE INSERT ON supply_contract
3  FOR EACH ROW
4  BEGIN
5      declare msg varchar(128);
6      if NEW.organization_contractor_id is not null
7      then
8          if not exists (
9                  SELECT 1
10                 FROM organization
11                 WHERE is_contractor = TRUE
12                 AND   organization.organization_id = NEW.organization_contractor_id
13                )
14         then
15             set msg = 'ERROR: ...[tg_supply_contract_i]';
16             signal sqlstate '45000' set message_text=msg;
17         end if;
18     end if;
19
20     if NEW.organization_customer_id is not null
21     then
22         if not exists (
23                 SELECT 1
24                 FROM organization
25                 WHERE is_corporate_customer = TRUE
26                 AND   organization.organization_id = NEW.organization_customer_id
27                )
28         then
29             set msg = 'ERROR: ...[tg_supply_contract_i]';
30             signal sqlstate '45000' set message_text=msg;
31         end if;
32     end if;
33
34     if NEW.person_id is not null
35     then
36         if not exists (
37                 SELECT 1
38                 FROM person
39                 WHERE is_personal_customer = TRUE
40                 AND   life_phase_enum = 'ADULT'
41                 AND   person.person_id = NEW.person_id
42                )
43         then
44             set msg = 'ERROR: ...[tg_supply_contract_i]';
45             signal sqlstate '45000' set message_text=msg;
46         end if;
47     end if;
48
49     if( SELECT  CASE WHEN NEW.organization_customer_id is null THEN 0 ELSE 1 END +
50                 CASE WHEN NEW.NEW.person_id is null is null THEN 0 ELSE 1 END
51       )  <> 1
52     then
53         set msg = 'ERROR: ...[tg_supply_contract_i]';
54         signal sqlstate '45000' set message_text=msg;
55     end if;
56 END; //
57 delimiter ;
```

Listing 2 Trigger for the `SUPPLY_CONTRACT` table.

A listing for the trigger of the `ORGANIZATION` table is provided in the Supplementary material.

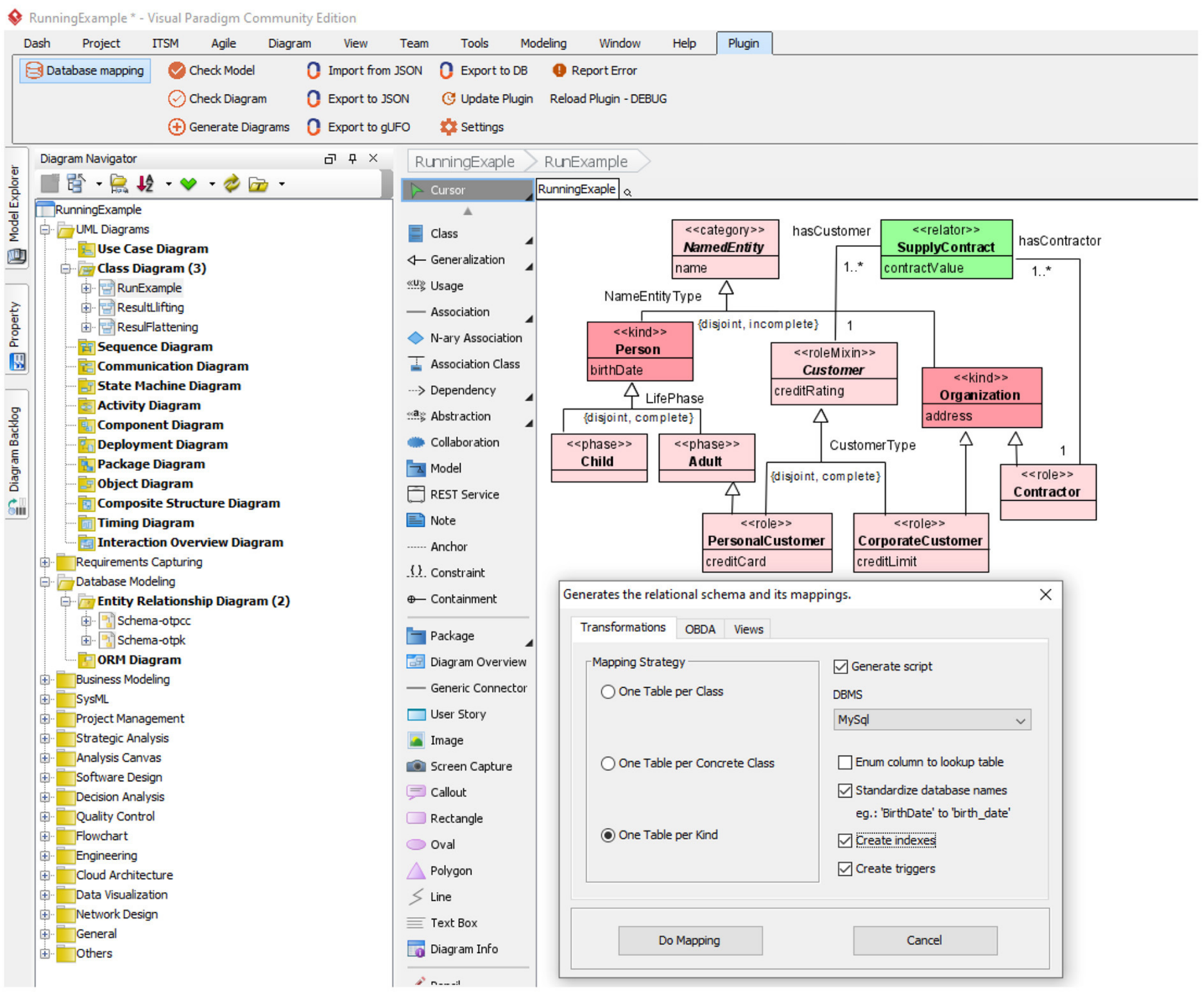
4.4. Implementation restrictions

Whenever associations have lower bound cardinality `1` in both association ends, there is a mutual dependency between the instances of the associated classes. This is the case in the example of the associations between `SupplyContract` and `Customer` and between `SupplyContract` and `Contractor`. In this case, enforcing both **MC5** and **MC6** after *each* insert becomes problematic, e.g., inserting a row in the `SUPPLY_CONTRACT` table would require an insertion in the `CUSTOMER` table and vice-versa. Unfortunately, this would require related inserts to be part of a single transaction, and a database implementation with transaction-level triggers, which is not the case here. Hence, we have opted not to enforce **MC5** and **MC6** in tandem in the implemented trigger generation. Violations of the missing constraint (**MC5** in this case) could be detected by a regular query that can easily be generated from the trace table following the same procedure to obtain the expressions concerning the triggering of **MC6**.

We have not yet implemented support for detecting violations of **MC7**, which applies to the emulation of specialization with keys in disjoint hierarchies. As discussed in Section 3.3, the constraint is not required in *one table per kind*, which has been the focus of our efforts. To support this constraint in general, further instrumentation of the transformation process is required, in particular due to flattening involving disjoint generalization sets. This is not necessary for the particular case of OntoUML models, since the required conditions for the relational schema can be derived directly from the source model by inspecting the classes stereotyped ≪kind≫ and ≪relator≫ along with generalization set constraints.

5. Implementation and tests

Figure 11 contains a screenshot of the implemented object-relational plugin for Visual Paradigm. The menu at the top includes the "Database mapping" button to access the



Generates the relational schema and its mappings.

Transformations | OBDA | Views

Mapping Strategy
- ○ One Table per Class
- ○ One Table per Concrete Class
- ● One Table per Kind

- ✓ Generate script
- DBMS: MySql
- ☐ Enum column to lookup table
- ✓ Standardize database names eg.: 'BirthDate' to 'birth_date'
- ✓ Create indexes
- ✓ Create triggers

FIGURE 11
A screenshot of the object-relational plugin interface.

functionalities of the plugin. (The other buttons displayed are part of the OntoUML plugin.[3]) The implementation supports three transformation approaches (*one table per class*, *one table per concrete class*, and *one table per kind*); different target DBMSs for the generated scripts (MySql, Postgres, and standard SQL 1999); and some optimization and customization parameters.

In addition to the formal specification and automated proofs for the flattening and lifting operations, we have performed a number of tests on the implemented transformation. We employed in the tests the first 30 projects published in the OntoUML repository (Barcelos et al., 2022), which are produced by third parties (with a few exceptions in which some of the authors of this paper were involved). We used the *one table per kind* approach, as it exercises both *lifting* and *flattening*. MySQL was selected as DBMS for the tests. All generated scripts and triggers are provided in the Supplementary material.

Table 4 provides a quantitative overview of the results, reporting the number of generated tables (excluding those used to represent `N:N` associations and multivalued attributes, which are created regardless of the chosen object-relational approach) as well as the number of constraints enforced by the resulting artifacts. Constraints `MC1` and `MC2` are reported together because the same generated command validates both cases (see, e.g., lines 50–56 of Listing 2). The same occurs for `MC3` and `MC4`. We excluded from the tests 8 of the first 30 projects due to (i) syntactic errors in the source models, (ii) presence of reserved keywords in labels, or (iii) complex handling of

3 OntoUML plugin was developed to facilitate the development of OntoUML models and verifies its compliance with OntoUML's syntactical rules (see Guizzardi et al., 2021 for more details), among other features. This plugin can be found at https://github.com/OntoUML/ontouml-vp-plugin.

TABLE 4 Transformation process results.

Project	Number of classes	Number of tables	MC1 and MC2 constraints	MC3 and MC4 constraints	MC6 constraint
aguiar2019ooco	55	19	12	16	28
ahmad2018aviation	21	5	0	10	16
aires2022valuenetworks-geo	24	19	18	0	6
amaral2019rot	48	26	48	2	18
amaral2020game-theory	22	11	0	2	16
amaral2020rome	48	35	38	0	16
amaral2022ethical-requirements	20	15	2	0	8
andersson2018value-ascription	13	11	0	0	2
bank-model	23	8	2	2	24
barcelos2013normative-acts	42	16	0	10	24
barros2020programming	12	1	0	6	0
Brazilian-governmental-organizational-structures	15	5	4	2	4
buchtela2020connection	19	6	0	0	2
buridan-ontology2021	53	20	0	10	82
carolla2014campus-management	17	12	0	0	18
castro2012cloudvulnerability	32	14	0	8	8
cgts2021sebim	29	10	0	2	2
chartered-service	11	11	0	0	0
clergy-ontology	29	13	0	14	58
cmpo2017	55	18	0	12	98
construction-model	13	7	0	4	6
debbech2019gosmo	22	10	0	2	34

datatypes (addressing complex datatype support and treating reserved keywords is not yet implemented in the prototype).

As expected, the transformation of some projects did not generate flattening-related `MC1` and `MC2` constraints because they do not have associations involving abstract classes. These constraints are quite numerous however in projects that include classes defined at a high level of generality (typical of "core ontologies") such as "aguiar2019oco," "aires2022valuenetworks-geo," "amaral2019rot," and "amaral2020rome." Further, some of the projects do not include `MC3` and `MC4` because they have no attributes in lifted classes or have no concrete class hierarchies of depth greater than two (in which discriminators from a previous lifting round are themselves lifted). `MC6` occurrences are the most common as they relate to the typing of associations in lifted classes (and thus are required for every association involving lifted classes independently of the cardinalities). The only projects with no occurrences of MC6 are "barros2020programming" (as it is a pure taxonomy with no associations) and "chartered-service" (that has associations, but does not employ specialization).

We have selected one project for exhaustive inspection, namely "aguiar2019ooco", which has the largest number of classes in the sample and for which all types of constraints were generated. We have manually inspected the source model to derive test cases. The test cases (reported in the Supplementary material) cover all of the constraints required to preserve the semantics of the source model in the present of lifting and flattening. Sixty test cases consist in attempting to insert or update data that would offend the original model semantics, and hence a pass in these test cases is an insert or update error raised by the generated triggers (the number of test cases does not correspond to the total number of missing constraints because some test cases check inserts only, some other test cases check updates only, and a number of those concern more than one missing constraint). The test cases jointly cover 100% of the conditions (IFs) in the generated triggers for the model. Another 36 test cases were added to insert or update data that does not offend the original model semantics, to make sure that they would not result in unintended insert or update errors.

An example of test case (`TC_AGUIAR_MC3and4_001`) is shown in Listing 3, that exercises the database corresponding to the fragment shown in Figure 12 (from "aguiar2019ooco"). As a result of the lifting process, two discriminator columns are introduced in the table corresponding to the `Language` kind. The `is_object_oriented_programming_language`

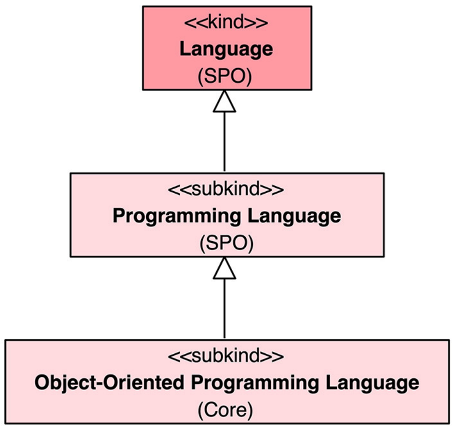


FIGURE 12
Fragment of the "aguiar2019ooco" project that results in iterated lifting.

discriminator column is introduced as a result of the first application of lifting, and the `is_programming_language` is introduced in the second application of lifting. As a consequence, the column `is_object_oriented_programming_language` can only be given a value when `is_programming_language = "1"` (i.e., corresponds to `true`). The test case attempts to insert a new language, with identifier 3 (`language_id = 3`), marked as an object-oriented programming language (`is_object_oriented_programming_language = "1"`), but not marking it as a programming language (`is_programming_language = "0"`). This should result in a violation of the insert trigger for the `language` table.

```
Test case name: TC_AGUIAR_MC3and4_001
Project name: aguiar2019ooco
Missing constraints evaluated: MC3and4          Number
     : 1          Action: Insert
Objective: Check whether MC3 and MC4 are correctly
     enforced in the case of lifting of "Object-
     Oriented Programming Language" and "Programming
     Language".
Test: Insert a row in the table 'language' that is
     marked as an "Object-Oriented Programming Language
     " but not as "Programming Language".
Expected result: Error.
Script:

INSERT INTO ‘language‘
       (‘language_id‘, ‘is_programming_language‘, ‘
     is_object_oriented_programming_language‘)
VALUES ('3', '0', '1');

Return message: ERROR: Violating conceptual model
     rules[tg_language_i].
Test result: PASS
```
Listing 3 Example test case `TC_AGUIAR_MC3and4_001`.

We have performed some simple tests to assess the performance overhead imposed with the introduction of the validation triggers. We selected the table from our test set that is accompanied by the largest number of constraints in triggers, namely the `METHOD_MEMBER_FUNCTION` table in "aguiar2019ooco." This table's triggers enforce seven constraints that resulted from lifting and flattening. We have contrasted the performance of 10 individual record inserts in the table with and without the triggers. We have found the following results: 10 ms on average for an insert in the table when the triggers are enabled; and 9 ms on average for an insert when the triggers are disabled. This indicates the overhead is not prohibitive. Further performance analysis in actual applications is straightforward, since triggers can be disabled and regular operation (that does not violate constraints) assessed directly.

Finally, in order to assess the performance of the transformation itself (design time performance), we have measured the time required to generate all relational schemas and triggers in the largest models in our test set. The results are presented in Table 5, showing the response times in seconds for the generation of the schema and of the scripts for the three largest projects in the test set. Measurements were obtained in a Windows 10 notebook with an i3 1.8 GHz processor (third generation), 250 GB SSD and 8 GB RAM.

TABLE 5 Transformation design time performance.

Project	Schema generation (time in seconds)	Triggers generation (time in seconds)
aguiar2019ooco	0.019	0.026
buridan-ontology2021	0.056	0.063
cmpo2017	0.168	0.237

6. Related work

Banerjee et al. (1987) propose well-defined rules that cover many aspects of object-oriented database schema manipulations, including "dropping an existing class." Differently from our work, their overall objective is to perform evolutionary manipulations of object-oriented schemas, so there is no concern for preservation of all aspects of the original model. The same can be said of several other works, like those of Penney and Stein (1987) and Lerner and Habermann (1990). They also rely on the solution space of object-oriented database systems.

Some work on refactoring strategies on UML class diagrams aim to preserve the syntactic correctness and/or semantics of the original model. For example, Markovic and Baar (2005) detail some refactoring rules along with their impacts on OCL constraints. Baar and Markovic (2006) focus on the preservation of semantics in the face of a "MoveAttribute" operation. As discussed in Section 3 concerning the refactoring presented by Fowler et al. (2012), the intent here is also *model editing*, and so the supported operations do not match those we require for relational schema realization.

There are also different approaches that aim to obtain SQL implementations of explicitly formulated OCL constraints. Some of these approaches, e.g., OCL2SQL (Demuth and Hußmann, 1999; Demuth et al., 2001), Incremental OCL constraints checking (Oriol and Teniente, 2014) and OCL_{FO} (Franconi et al., 2014) are focused on mapping OCL Boolean expressions only, while others such as SQL-PL4OCL (Egea and Dania, 2017) and MySQL4OCL (Egea et al., 2010) consider OCL expressions in general. One other approach, namely, GeoUML (Pelagatti et al., 2009) considers the SQL implementation of special-purpose OCL constraints for expressing geo-spatial relations.

Some ontology-based approaches focus on: (i) the relational realization of computational OWL ontologies, or (ii) on the access of data in relational databases using computational ontologies. In the former group, we have approaches such as those of Vyšniauskas et al. (2011) and Afzal et al. (2016), which implement the *one table per class* strategy, thus, mapping each OWL class to a relational table. In the latter group, we have Ontology-Based Data Access (OBDA) techniques (Poggi et al., 2008) that work with the translation of high-level queries into SQL queries. Users of an OBDA solution are required to write a mapping specification that establishes how entities represented in the relational database should be mapped to instances of classes in a computational (RDF- or OWL-based) ontology. The OBDA solution then enables the expression of queries in terms of the ontology, e.g., using SPARQL. Each query is rewritten by the OBDA solution following the manually-written mappings into SQL queries that are executed at the database. Results of the query are then mapped back to triples and consumed by the user using the vocabulary established at the ontology. See Guidoni et al. (2021) for an integration of ODBA techniques with the forward engineering of relational schemas as described in this paper. In that work, we provide an approach to automatically generate the otherwise manually-rewritten mappings. These mappings are used by the ODBA solution to automatically rewrite queries expressed in terms of the source conceptual model. This includes the automatic rewriting of polymorphic queries expressed in terms of superclasses that are flattened out.

There is a variety of studies focusing on mapping structural conceptual models, such as EER diagrams or UML class diagrams, into relational schemas, such as those performed by Shah and Slaughter (2003), Hull and King (1987), Teorey et al. (1986), and Pergl et al. (2013). Shah and Slaughter (2003) discuss the various class-to-table strategies but do not provide detailed model transformation rules for realizing these strategies, and do not consider the consequences of the strategies in terms of preservation of semantics. The seminal study presented by Teorey et al. (1986) proposes transformation rules to bridge the constructs of EER diagrams with those of relational schemas, identifying missing constraints for disjoint generalizations as we discussed in Section 3.3. More recently, Pergl et al. (2013) have discussed how to transform OntoUML models into relational schemas. In his Ph.D. thesis, Rybola (2017) proposes the transformation process, presenting quite sophisticated ways of preserving integrity constraints. Validation triggers were proposed to preserve the semantics of the original constructs from the source OntoUML model (*modal aspects* of the stereotypes which we do not address here). A common trait of these approaches is that they consider solely the *one table per class* strategy for transforming generalization hierarchies. While this yields a straightforward relation between the conceptual model and the relational schema, using the resulting relational schema may be cumbersome depending on the source conceptual model, in particular in the presence of generalization hierarchies, which are emulated as we discussed in Section 3.3. For example, retrieving all the attributes of an instance of a leaf class requires joins involving a number tables equal to the depth of the whole specialization hierarchy. Because of this, these tools are usually employed by providing a UML class diagram that is in fact a visual representation of the relational schema: the model is produced manually at a lower level of abstraction, with automation restricted to straightforward translation. In some cases, stereotypes for primary and foreign keys are introduced in the notation (Shah and Slaughter, 2003).

Alloy is commonly used in the literature to validate semantic changes in UML class models. For example, the work of Cunha et al. (2015) proposes to identify semantic losses through bidirectional transformations using Alloy. Their job consists of performing transformations for Alloy through the UML and OCL specifications, and to translate them back to UML and OCL, thus allowing to verify and validate the results of a transformation in Alloy. Gheyi et al. (2007) present a catalog of primitive transformations to predict whether a change in the source model will cause semantic loss. To do so, the authors formalize a static semantics for Alloy to match the source model semantics. With this, it is possible to predict whether any change in the source model will cause semantic losses. These approaches provide useful general frameworks which could be employed to formalize the operations we have discussed here.

Finally, there are a number of model transformation specification languages in the literature, including most prominently ATL (Jouault and Kurtev, 2005), the Epsilon Transformation Language (ETL) (Kolovos et al., 2008) and those solutions implementing the MOF QVT specification (Kurtev, 2007). These languages focus on the *specification* of a model transformation in terms of the constructs present in source and target metamodels. Model transformation specifications written in these languages are usually interpreted with a corresponding transformation engine to execute the transformation of a specific source model. Model transformation languages operate at the abstract syntax level and are neutral with respect to the semantics of the source and target languages. We focus here instead on the *content* of the transformation, i.e., we are concerned with the design decisions that can be generalized into a specific model transformation and with the semantics of the source model and its correspondence with the resulting relational schema. In principle, any of these transformation languages could have been used for the implementation of the transformation we present here.

7. Conclusions

We have focused here on the consequences of various class-to-table transformation strategies, such that the relational schema can be complemented with integrity constraints that reflect the source conceptual model constraints. We have formulated the approach in terms of the consequences of flattening and lifting thus account for various transformation strategies in the literature, including *one table per concrete class*, *one table per leaf class*, *one table per kind*, *one table per hierarchy*.

A common characteristic of many approaches in the literature (Teorey et al., 1986; Hull and King, 1987; Shah and Slaughter, 2003; Pergl et al., 2013) is that they consider solely the *one table per class* strategy for transforming generalization hierarchies, which is also the approach adopted in the various commercial tools to perform this kind of transformation. A result is that the conceptual model is in fact produced manually by the modeler at a lower level of abstraction, with automation restricted to straightforward translation. That is detrimental to the objectives of model-driven development, including abstraction and platform-independence. In this light, our work contributes to a fuller application of model-driven transformation of conceptual models, by effectively decoupling conceptual models from their implementation.

We have elaborated a formalization (see Supplementary material) based on a simple translational semantics to demonstrate that the application of the operations of flattening and lifting does not introduce elements that contradict the original source model (axioms added due to flattening and lifting do not lead to an unsatisfiable theory). Further, the formalization we have provided demonstrates that the missing constraints are indeed required to preserve constraints implied by the model before the application of each operation. Despite that, more general formalization would be desirable. In particular, we have yet to demonstrate formally that the set of missing constraints we have identified is *complete* with respect to the original model.

Approaches that use flattening require polymorphic queries to be written with unions of the (base) tables corresponding to the subclasses of the flattened class. For example, queries involving instances of `Customer` in our running example must be written in terms of the union of `PERSONAL_CUSTOMER` and `CORPORATE_CUSTOMER` in the *one table per concrete class* approach. We have already worked out an approach to be reported soon based on the generation of database views that produce the unions that are required for polymorphic queries involving classes that are flattened. This simplifies polymorphic query rewriting, and reflects flattened classes at the level of the relational schema. This approach also leverages the trace sets we have discussed in this paper. See Guidoni et al. (2021) for a solution for automatic polymorphic query rewriting based on OBDA techniques integrated with the approach described in this paper.

Approaches based on lifting further require the user to perform queries involving instances of lifted subclasses to be written with filters based on the generated discriminators. This is also addressed in our view generation approach, and is supported by the OBDA-based approach in Guidoni et al. (2021).

An important limitation of the approach we discuss here is that we do not deal with existing (OCL) constraints that need to be rewritten in the transformation process due to lifting and flattening. (For example, consider the constraint that a contractor should not be a customer of itself in the scope of a supply contract). We expect we can profit from query rewriting strategies in the literature—such as those used in Ontology-Based Data Access (Calvanese et al., 2017)—to address this in the future. We also aim to investigate to what extent we can leverage the aforementioned views such that OCL invariants can be directly enforced at the database level with references to those views instead of rewriting. We also do not address behavioral and dynamic aspects, and provide no special treatment for whole-part relations.

Data availability statement

The original contributions presented in the study are included in the article/Supplementary material, further inquiries can be directed to the corresponding author/s.

Author contributions

GLG implemented the transformation and performed the tests. JPAA contributed with the formalization. All authors contributed to the conception of the approach, manuscript revision, read, and approved the submitted version.

Funding

This research was partly funded by Brazilian funding agencies CNPq (grants numbers 313687/2020-0 and 407235/2017-5) and CAPES (grant number 23038.028816/2016-41).

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fcomp.2022.1020168/full#supplementary-material

References

Afzal, H., Waqas, M., and Naz, T. (2016). OWLMap: fully automatic mapping of ontology into relational database schema. *Int. J. Adv. Comput. Sci. Appl.* 7, 1–15. doi: 10.14569/IJACSA.2016.071102

Ambler, S. W. (2003). *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York, NY: Wiley.

Baar, T., and Markovic, S. (2006). "A graphical approach to prove the semantic preservation of UML/OCL refactoring rules," in *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Vol. 4378 of Lecture Notes in Computer Science* (Novosibirsk: Springer), 70–83.

Banerjee, J., Kim, W., Kim, H., and Korth, H. F. (1987). "Semantics and implementation of schema evolution in object-oriented databases," in *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference* (San Francisco, CA: ACM Press), 311–322. doi: 10.1145/38713.38748

Barcelos, P. P. F., Sales, T. P., Fumagalli, M., Fonseca, C. M., Sousa, I. V., Romanenko, E., et al. (2022). "A fair model catalog for ontology-driven conceptual modeling research," in *41st International Conference Proceedings (ER 2022)*, 17–20. Available online at: https://purl.org/ontouml-models/ doi: 10.1007/978-3-031-17995-2_1

Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., et al. (2017). Ontop: answering SPARQL queries over relational databases. *Semant. Web* 8, 471–487. doi: 10.3233/SW-160217

Cunha, A., Garis, A. G., and Riesco, D. (2015). Translating between alloy specifications and UML class diagrams annotated with OCL. *Softw. Syst. Model.* 14, 5–25. doi: 10.1007/s10270-013-0353-5

Demuth, B., and Hußmann, H. (1999). "Using UML/OCL constraints for relational database design," in *Proceedings of UML'99, Vol. 1723 of LNCS* (Berlin; Heidelberg: Springer), 598–613. doi: 10.1007/3-540-46852-8_42

Demuth, B., Hußmann, H., and Loecher, S. (2001). "OCL as a specification language for business rules in database applications," in *UML 2001—The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Vol. 2185 of Lecture Notes in Computer Science*, eds M. Gogolla and C. Kobryn (Toronto, ON: Springer), 104–117. doi: 10.1007/3-540-45441-1_9

Egea, M., and Dania, C. (2017). "SQL-PL4OCL: an automatic code generator from OCL to SQL procedural language," in *Proceedings of MODELS'17* (Austin, TX), 54. doi: 10.1109/MODELS.2017.34

Egea, M., Dania, C., and Clavel, M. (2010). MySQL4ocl: a stored procedure-based MySQL code generator for OCL. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 36.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (2012). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Pearson Education.

Franconi, E., Mosca, A., Oriol, X., Rull, G., and Teniente, E. (2014). "Logic foundations of the OCL modelling language," in *Proceedings of 14th European Conference on Logics in Artificial Intelligence (JELIA), Vol. 8761*, 657–664. doi: 10.1007/978-3-319-11558-0_49

Gheyi, R., Massoni, T., and Borba, P. (2007). A static semantics for alloy and its impact in refactorings. *Electron. Notes Theor. Comput. Sci.* 184, 209–233. doi: 10.1016/j.entcs.2007.03.023

Guidoni, G. L., Almeida, J. P. A., and Guizzardi, G. (2020). "Transformation of ontology-based conceptual models into relational schemas," in *Proceedings of ER 2020, Vol. 12400 of Lecture Notes in Computer Science* (Springer), 315–330. doi: 10.1007/978-3-030-62522-1_23

Guidoni, G. L., Almeida, J. P. A., and Guizzardi, G. (2021). "Forward engineering relational schemas and high-level data access from conceptual models," in *40th International Conference on Conceptual Modeling (ER 2021), Vol. 13011 of Lecture Notes in Computer Science* (Springer), 133–148. doi: 10.1007/978-3-030-89022-3_12

Guizzardi, G. (2005). *Ontological foundations for structural conceptual models* (Ph.D. thesis). University of Twente, Enschede, Netherlands.

Guizzardi, G., Figueiredo, G., Hedblom, M. M., and Poels, G. (2019). "Ontology-based model abstraction," in *13th International Conference on Research Challenges in Information Science, RCIS 2019* (Brussels), 1–13. doi: 10.1109/RCIS.2019.8876971

Guizzardi, G., Fonseca, C. M., Almeida, J. P. A., Sales, T. P., Benevides, A. B., and Porello, D. (2021). Types and taxonomic structures in conceptual modeling: a novel ontological theory and engineering support. *Data Knowl. Eng.* 134:101891. doi: 10.1016/j.datak.2021.101891

Guizzardi, G., Fonseca, C. M., Benevides, A. B., Almeida, J. P. A., Porello, D., and Sales, T. P. (2018). "Endurant types in ontology-driven conceptual modeling: towards OntoUML 2.0," in *Conceptual Modeling - 37th International Conference, ER 2018, Proceedings, Vol. 11157 of Lecture Notes in Computer Science* (Springer), 136–150. doi: 10.1007/978-3-030-00847-5_12

Hull, R., and King, R. (1987). Semantic database modeling: survey, applications, and research issues. *ACM Comput. Surv.* 19, 201–260. doi: 10.1145/45072.45073

Iacob, M. E., Steen, M. W., and Heerink, L. (2008). "Reusable model transformation patterns," in *2008 12th Enterprise Distributed Object Computing Conference Workshops*, 1–10. doi: 10.1109/EDOCW.2008.51

Jouault, F., and Kurtev, I. (2005). "Transforming models with ATL," in *International Conference on Model Driven Engineering Languages and Systems* (Springer), 128–138. doi: 10.1007/11663430_14

Keller, W. (1997). "Mapping objects to tables: a pattern language," in *EuroPLoP 1997: Proceedings of 2nd European Conference Pattern Languages of Programs* (Kloster Irsee: Siemens Tech), 207.

Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2008). "The epsilon transformation language," in *Theory and Practice of Model Transformations*, eds A. Vallecillo, J. Gray, and A. Pierantonio (Berlin; Heidelberg: Springer Berlin Heidelberg), 46–60. doi: 10.1007/978-3-540-69927-9_4

Kurtev, I. (2007). "State of the art of QVT: a model transformation language standard," in *International Symposium on Applications of Graph Transformations with Industrial Relevance* (Springer), 377–393. doi: 10.1007/978-3-540-89020-1_26

Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., and Sharbaf, M. (2018). A survey of model transformation design patterns in practice. *J. Syst. Softw.* 140, 48–73. doi: 10.1016/j.jss.2018.03.001

Lerner, B. S., and Habermann, A. N. (1990). "Beyond schema evolution to database reorganization," in *Conference on Object-Oriented Programming*

Systems, Languages, and Applications/European Conference on Object-Oriented Programming, OOPSLA/ECOOP 1990, ed A. Yonezawa (Ottawa, ON: ACM), 67–76. doi: 10.1145/97945.97956

Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G. M. K., et al. (2016). Model transformation intents and their properties. *Softw. Syst. Model.* 15, 647–684. doi: 10.1007/s10270-014-0429-x

Markovic, S., and Baar, T. (2005). "Refactoring OCL annotated UML class diagrams," in *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Proceedings, Vol. 3713 of Lecture Notes in Computer Science* (Montego Bay: Springer), 280–294. doi: 10.1007/11557432_21

Markowitz, V. M., and Shoshani, A. (1992). Representing extended entity-relationship structures in relational databases: a modular approach. *ACM Trans. Database Syst.* 17, 423–464. doi: 10.1145/132271.132273

Oriol, X., and Teniente, E. (2014). "Incremental checking of OCL constraints through SQL queries," in *Proceedings of MODELS 2014, Vol. 1285 of CEUR Workshop Proceedings*. doi: 10.1007/978-3-319-25264-3_15

Pelagatti, G., Negri, M., Belussi, A., and Migliorini, S. (2009). "From the conceptual design of spatial constraints to their implementation in real systems," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS '09* (New York, NY: Association for Computing Machinery), 448–451. doi: 10.1145/1653771.1653841

Penney, D. J., and Stein, J. (1987). "Class modification in the gemstone object-oriented DBMS," in *Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1987* (Orlando, FL: ACM), 111. doi: 10.1145/38765.38817

Pergl, R., Sales, T. P., and Rybola, Z. (2013). "Towards OntoUML for software engineering: from domain ontology to implementation model," in *Model and Data Engineering - Third International Conference, MEDI 2013, Proceedings, Vol. 8216 of Lecture Notes in Computer Science* (Springer), 249–263. doi: 10.1007/978-3-642-41366-7_21

Philippi, S. (2005). Model driven generation and testing of object-relational mappings. *J. Syst. Softw.* 77, 193–207. doi: 10.1016/j.jss.2004.07.252

Poggi, A., Lembo, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., and Rosati, R. (2008). Linking data to ontologies. *J. Data Semant.* 10, 133–173. doi: 10.1007/978-3-540-77688-8_5

Rybola, Z. (2017). *Towards OntoUML for software engineering: transformation of OntoUML into relational databases* (Ph.D. thesis). Czech Technical University, Prague, Czechia. doi: 10.15439/2016F250

Shah, D., and Slaughter, S. (2003). "Transforming UML class diagrams into relational data models," in *UML and the Unified Process* (IGI Global), 217–236. doi: 10.4018/978-1-93177-744-5.ch010

Teorey, T. J., Yang, D., and Fry, J. P. (1986). A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput. Surveys* 18, 197–222. doi: 10.1145/7474.7475

Torres, A., Galante, R., Pimenta, M. S., and Martins, A. J. B. (2017). Twenty years of object-relational mapping: a survey on patterns, solutions, and their implications on application design. *Inform. Softw. Technol.* 82, 1–18. doi: 10.1016/j.infsof.2016.09.009

Vyšniauskas, E., Nemuraitt, L., Butleris, R., and Paradauskas, B. (2011). Reversible lossless transformation from owl 2 ontologies into relational databases. *Inform. Technol. Control* 40, 293–306. doi: 10.5755/j01.itc.40.4.979