Software and Systems Modeling (2022) 21:587–621
https://doi.org/10.1007/s10270-022-00973-7

THEME SECTION PAPER

Domain object hierarchies inducing multi-level models

Bernd Neumayr[1] · **Michael Schrefl**[1]

Received: 25 May 2020 / Revised: 30 November 2021 / Accepted: 3 January 2022 / Published online: 11 March 2022

Abstract
Conceptual modeling of domain object hierarchies, such as product hierarchies or organization hierarchies, is difficult due to the intricate nature of nonphysical domain objects organized in such hierarchies. Modeling domain object hierarchies as part-whole hierarchies covers their hierarchical structure, yet to capture their meaning, part-whole hierarchies have to be combined with specialization and multi-level instantiation. To this end we introduce the *deep domain object* (DDO) multi-level modeling pattern and approach. With the DDO approach, subclasses and metaclasses are induced by and integrated with the part-whole hierarchy. The approach is aligned with the multi-level theory (MLT) and formalized by a metamodel and a set of deductive rules implemented in F-Logic. The proof-of-concept prototype is used for automated application of the pattern and for querying induced multi-level models.

Keywords Conceptual modeling · Multi-level modeling · Metamodeling · Part-whole · Generalization · Abstraction · Concretization

1 Introduction

Conceptual modeling is the activity of formalizing some aspects of the physical and social world around us for purposes of understanding and communication [26,33], typically in the context of information systems development. With regard to databases, conceptual modeling aims at 'capturing the meaning of a database' [26] independent of its logical and physical implementation.

Before we discuss the aims and contributions of this paper, let us first explain and illustrate our notion of domain object hierarchy. While the approach to modeling domain object hierarchies investigated in this paper is applicable to structural conceptual modeling in general, it is particularly geared towards the conceptual modeling of enterprise databases underlying enterprise information systems, such as ERP and CRM systems.

Enterprise information systems deal, to a large extent, with social worlds where many of the *domain objects*, i.e., the relevant entities from the world represented in the enterprise database, are *nonphysical objects* [13], like sales divisions or product categories.

Domain objects may be organized in hierarchies where domain objects at higher levels not only but also act as abstractions of their subordinate objects. For example, sales employees, sales outlets, and sales divisions are organized in a sales organization hierarchy with a sales division also acting as abstraction of its subordinate sales employees and sales outlets. Likewise, product individuals, product models and product categories are arranged in a product hierarchy, with a product category also acting as abstraction of its subordinate product models and product individuals.

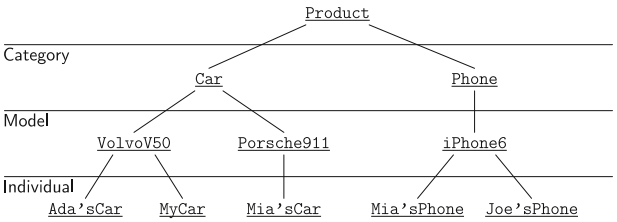
To make these abstraction roles of domain objects more tangible, we say a higher-level domain object induces a class for each subordinate level. Figure 1b shows the classes together with their extensions induced in this way by the product hierarchy of Fig. 1a. For example, a higher level domain object such as `Car` at level Category induces classes such as `CarModel` and `CarIndividual`, which have the sets of subordinate domain objects at levels Model and `Individual`, respectively, as extensions.

With these induced classes and their extensions, the instance-of (set membership) and subclass-of (subset) relationships, which are implicit in the domain object hierarchy, become tangible. For example, `VolvoV50` is instance of induced class `CarModel`, i.e., it is member of the extension of `CarModel`. The latter, in turn, is a subclass of class

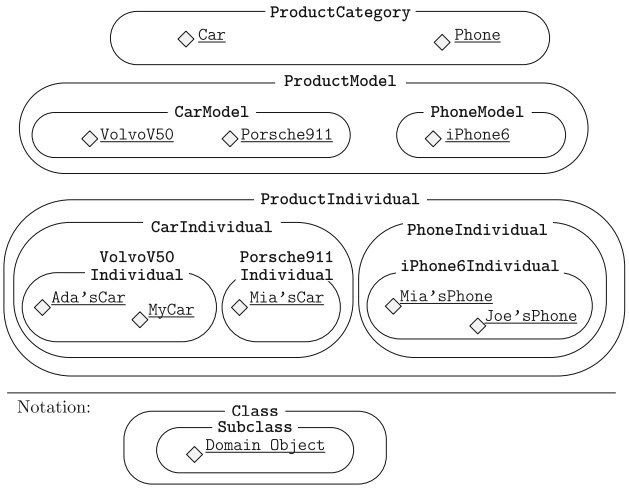
Communicated by Adrian Rutle and Manuel Wimmer.

✉ Bernd Neumayr
bernd.neumayr@jku.at

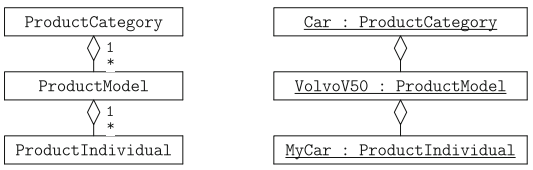
[1] Johannes Kepler University Linz, Linz, Austria



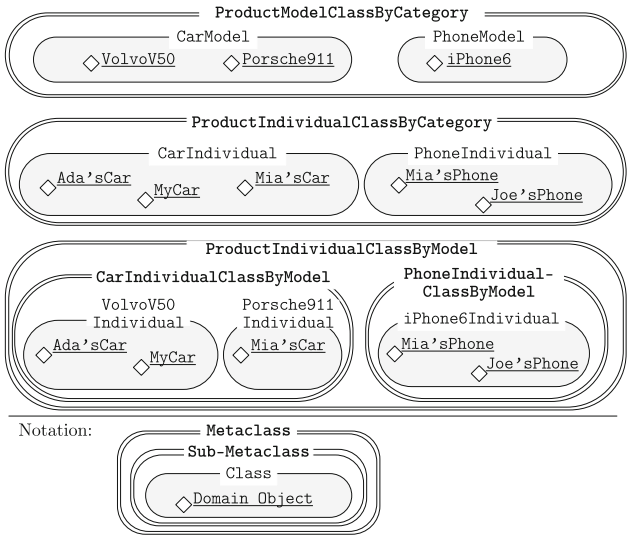
(a) Informal representation of a domain object hierarchy



(b) Induced classes with their extensions (sets of domain objects), e.g., `VolvoV50` is an instance of `CarModel` which is a subclass/subset of `ProductModel`



(c) Representing the hierarchical structure of a domain object hierarchy as part-whole hierarchy in UML



(d) Induced metaclasses with their extensions (sets of classes), e.g., class `VolvoV50Individual` is an instance of metaclass `CarIndividualClassByModel` which is a subclass of `ProductIndividualClassByModel`

Fig. 1 A sample domain object hierarchy inducing classes and metaclasses

`ProductModel`, that is, its extension is a subset of the extension of `ProductModel`.

Let us now turn to the metaclasses implicit in a domain object hierarchy. As illustrated in Fig. 1d, induced classes can be collected into metaclasses with regard to the level of their instances and with regard to the class of the domain object by which they are induced. For example, classes `CarModel` and `PhoneModel` have instances at level Model and are each induced by a member of class `ProductCategory`. Hence, they can be collected into an induced metaclass `ProductModelClassByCategory`.

What we *aim* for in this paper is a conceptual modeling approach for representing such hierarchies. The approach should make explicit the induced classes and metaclasses so that the modeler can use them for specialization and information aggregation. We are especially interested in supporting the following modeling challenges: (1) *specialization* along the hierarchy, e.g., product models have a list price, car models additionally have a maximum speed. (2) *Regulating specialization* along the hierarchy, e.g., every product model is associated with a sales employee acting as product manager. By linking a product category to a sales division, the range of property product manager is refined to employees of that sales division. (3) *Information aggregation* along the hierarchy, e.g., currently, the average list price of car models is € 21,034.

Based on an initial representation of domain object hierarchies as part-whole hierarchies in UML with levels represented as classes and domain objects as instances of these classes, see Fig. 1c, the paper makes the following *contributions*: (1) the *deep domain object (DDO) pattern* is a multi-level modeling pattern for extending domain object hierarchies which are modeled as part-whole hierarchies with induced subclasses and induced metaclasses. The DDO pattern is related to 'promotion with base classes' [24], a solution to the type-object pattern, and is aligned with the multi-level theory (MLT) [8]. (2) The *DDO approach* is a multi-level modeling approach based on the automatic and recursive application of the DDO pattern together with mechanisms for specialization based on induced subclasses, regulating specialization based on induced metaclasses, and aggregated information provided with classes and metaclasses. (3) A proof-of-concept prototype in F-Logic which can be used to

query the induced multi-level model and which is available open-source.

In this paper we introduce the DDO approach on top of a simplified multi-level modeling language. The main simplification is that all properties are single-valued, mandatory and uni-directional. While the DDO pattern is independent of these simplifications, they facilitate an in-depth discussion of specialization, regulating specialization, as well as information aggregation along the hierarchy, without getting drowned by the complexities of optional, multi-valued, or bi-directional properties.

The remainder of this paper is structured as follows. Section 2 introduces a running example consisting of several domain object hierarchies modeled as part-whole hierarchies. Section 3 extends the running example with modeling challenges together with a baseline solution in UML. Section 4 discusses relevant aspects of MLT and Dual Deep Modeling (DDM) [30], which provided the starting point for the development of the DDO approach. Section 5 introduces a set of deductive rules for deriving a multi-level model from a part-whole hierarchy by, in principle, recursive application of the DDO pattern. Section 6 demonstrates the use of deep domain objects for solving the modeling challenges. Section 7 discusses the realization of the DDO approach in F-Logic and demonstrates its use for querying the induced multi-level model, with the complete code provided in Appendix A together with the fully-worked running example in Appendix B. Sect. 8 gives an overview of related work. Section 9 concludes the paper.

2 Starting Point: Domain Object Hierarchies modeled as Part-Whole Hierarchies

In this section we describe and exemplify the two-level model that serves as core of the running example throughout the paper.

Global two-level model. The starting point for conceptual modeling with domain object hierarchies is a two-level model (depicted in Fig. 2) comprising a schema level (as UML class diagram) and an instance level (as UML object diagram). The modeler has already identified three domain object hierarchies, the Product hierarchy, the Sales organisation hierarchy, and the Engine hierarchy, marked with blue, green, and red background, respectively.

Note that there may be classes in the global model that are not part of a domain object hierarchy, for example the class `Customer`.

Domain object hierarchy schema. At the schema level, a domain object hierarchy gets a name and comprises a set of classes that represent the levels of the hierarchy. Each class in a domain object hierarchy gets a level name which is unique within the hierarchy. The set of classes (and hence levels) in a domain object hierarchy is arranged in a path (e.g., the Product hierarchy), a tree (e.g., the Sales hierarchy), or a forest (no example given) by aggregation relationships. When we say a class or level is *under* another class or level, respectively, we refer to the transitive closure of the aggregation relationships.

For example, the Sales domain object hierarchy comprises classes `SalesDivision`, `SalesEmployee`, and `SalesOutlet` which represent hierarchy levels Division, Employee, and Outlet, respectively. These classes are connected by aggregation relationships expressing that each sales outlet and each sales employee belongs to exactly one sales division. `SalesEmployee` and `SalesOutlet` are under `SalesDivision`.

The Product domain object hierarchy comprises classes `ProductCategory`, `ProductModel`, and `ProductIndividual` which represent hierarchy levels Category, Model, and Individual. These classes are connected by aggregation relationships expressing that each product individual belongs to some product model which in turn belongs to some product category. `ProductIndividual` is under `ProductModel` and under `ProductCategory`, `ProductModel` is under `ProductCategory`.

Properties. A class specifies the schema of its member objects by a set of properties. Each property has as range a class or a simple datatype. For simplicity's sake we only consider properties that are uni-directional, mandatory and functional.

For example, the class `ProductCategory` defines a property `vatRate` with range `Number` and property `salesDivision` with range `SalesDivision`. Class `ProductModel` defines properties `vatRate` and `listPrice` with range `Number` and property `productMgr` with range `SalesEmployee`. Class `ProductIndividual` defines properties `vatRate` and `soldPrice` with range `Number`, property `soldTo` with range `Customer`, and property `soldAt` with range `SalesOutlet`.

Domain object hierarchy instance. At the instance level, a domain object hierarchy is represented by a part-whole hierarchy obeying the hierarchy schema. Every domain object has at most one *parent* object in the part-whole hierarchy. A domain object o is *descendant* of domain object o' if o is modeled as a direct or indirect part of o'. A domain object o *is at level* k if o is an instance of class c which represents level k.

For example, domain object `Car` is the parent of domain objects `VolvoV50` and `Porsche911`. `VolvoV50` is the parent of `Ada'sCar` and `MyCar`. `VolvoV50` and `Porsche911` are the descendants of `Car` at level Model and `Ada'sCar`, `MyCar`, and `Mia'sCar` are the descendants of `Car` at level Individual.

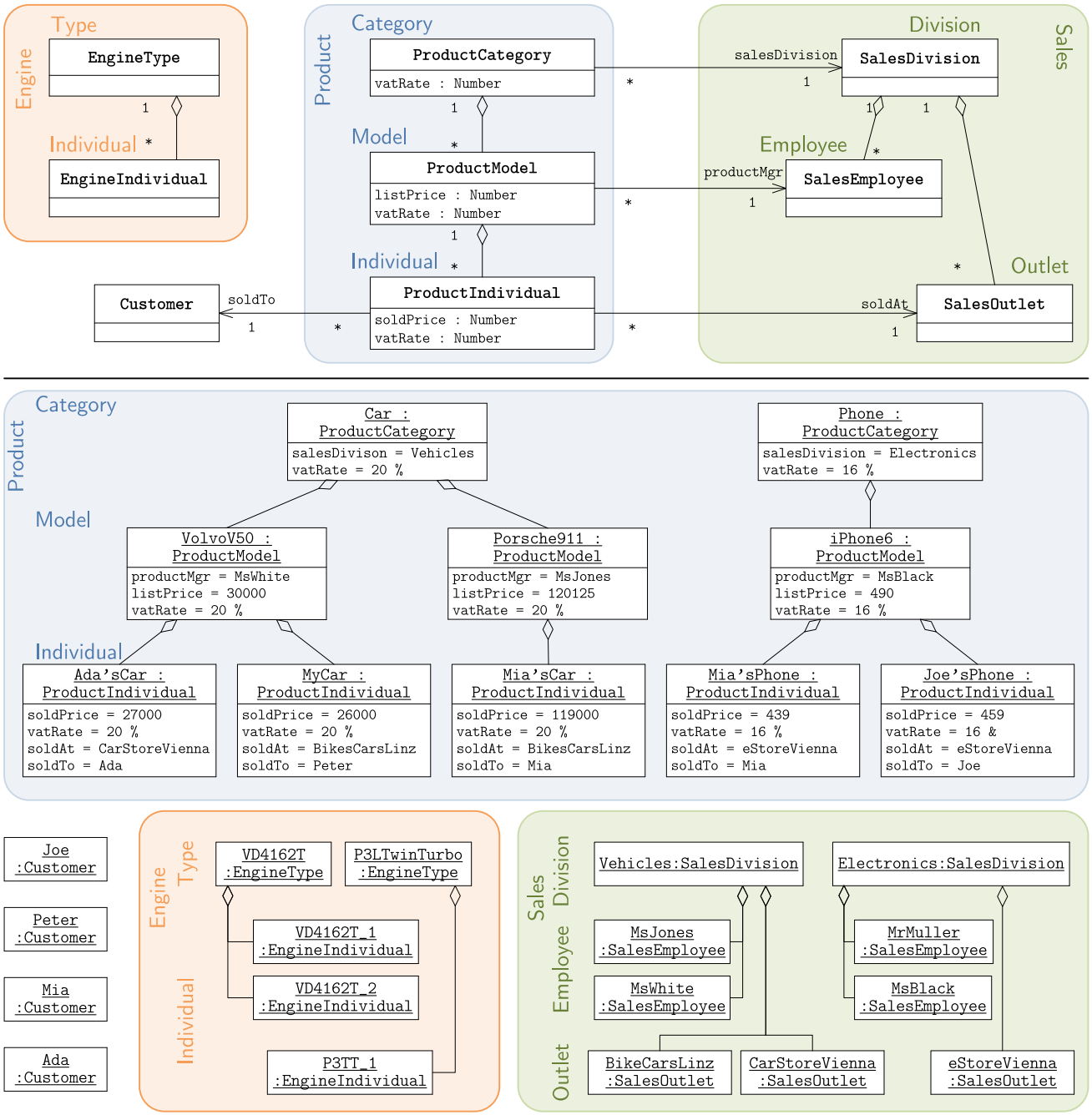


Fig. 2 Sample domain object hierarchies modeled as part-whole hierarchies in UML. The modeler identifies and gives names to domain object hierarchies by annotating them in the class diagram, classes that are part of a domain object hierarchy get a level name which is unique within their hierarchy. For illustration purposes, also the object diagram is annotated with hierarchy and level names

Domain objects assign a value to each property introduced with their class, obeying the property's range. For example, individual product `MyCar` of product model `VolvoV50` was sold for € 26.000 at sales outlet `BikesCarsLinz` to customer `Peter` with a VAT rate of 20%. Product model `VolvoV50` of product category `Car` has a list price of € 30.000, a VAT rate of 20% and has sales employee `MsWhite` as product manager. Product category `Car` has a VAT rate of 20% and is distributed by the `Vehicles` sales division. Sales outlet `BikesCarsLinz` and sales employee `MsWhite` both belong to the `Vehicles` sales division.

3 Modeling challenges: specialization and information aggregation along the hierarchy

In this section we break down the three modeling challenges associated with domain object hierarchies into specific modeling tasks. Preliminary modeling solutions in UML (with extensions) are provided as baseline for the qualitative evaluation of the DDO approach.

3.1 Desiderata

Before we introduce the modeling challenges let us discuss desiderata for modeling solutions.

- *Non-disruptive:* modeling domain object hierarchies as part-whole hierarchies of physical and mainly nonphysical objects is a pragmatic modeling choice reflecting how such hierarchies are often represented in enterprise databases. Modeling solutions should be non-disruptive in that they should not require the replacement of the existing conceptual model but act as an add-on to the part-whole hierarchy.
- *Conceptual clarity:* a domain object in a domain object hierarchy not only represents itself but also acts as an abstraction of its descendants at various levels. Conceptual clarity can be reached by clearly indicating with a property whether it describes (i) the domain object as such, (ii) commonalities of descendants at a particular level, or (iii) aggregated information about the set of descendants at a particular level.
- *Compactness* (modularity, cf. [29] and scoping): All model elements that are introduced with a domain object to describe the domain object itself or to characterize its descendants at various levels should be tied together and it should be possible to use and update them as a self-contained part of the model without interfering with the rest of the model.
- *Factoring out commonalities:* a modeling solution should factor out commonalities as much as possible to avoid the manual duplication of the same modeling information with various model elements. The modeler should not be forced to model by hand additional classes and additional dependencies that redundantly represent the hierarchical relationships in the domain object hierarchy. The modeler should, for example, not be forced to introduce the same aggregated property multiple times.
- *Avoid custom constraints:* modeling challenges should not be solved by custom constraints in natural language, OCL, or similar, but by applying adequate modeling constructs which serve much better the purposes of conceptual modeling, namely understanding and communication, and are less error-prone in development and maintenance.

3.2 Specialization along the hierarchy

When adding a new domain object, e.g., a new product category, it should be possible to specialize the schema for descendant domain objects at various hierarchy levels. We will, first, describe the modeling tasks and exemplify them for product category `Car` in the running example. We will then present a preliminary solution in UML and then discuss the quality of the solution with regard to the desiderata (see Sect. 3.1).

The to-be supported modeling tasks are:

C1.1 *Range refinement* The modeler should be able to refine for the descendants of a domain object o at level k the range of a property p to descendants of a domain object o' at level k', e.g.:

- For car models, i.e., descendants of `Car` at level Model, the range of `productMgr` is refined to descendants of sales division `Vehicles` at level Employee.
- For car individuals, i.e., descendants of `Car` at level Individual, the range of `soldAt` is refined to descendants of sales organisation `Vehicles` at level Outlet.

C1.2 *Extending the schema* The modeler should be able to extend the schema of descendants of a domain object o at level k by introducing additional properties and specify as range a class from the global model, or the set of descendants of a domain object o' at level k', or a simple datatype, e.g.:

- Descendants of `Car` at level Model have an additional property `engineType` with range `Engine Type` (from the Engine hierarchy).
- Car individuals, i.e., product individuals of category `Car`, are associated with an individual engine which is of a certain engine type.

C1.3 *Shared values* The modeler should be able to specify for a property p a value v that is shared by all descendants of a domain object o at level k, e.g.:

- The VAT rate of all car models is 20%.
- The VAT rate of all car individuals is 20%.

Preliminary solution. Our preliminary solution (Fig. 3) uses UML with derived classes [14] and shared property values. Derived classes are marked with / and the derivation condition is given in natural language. Shared values, i.e., property

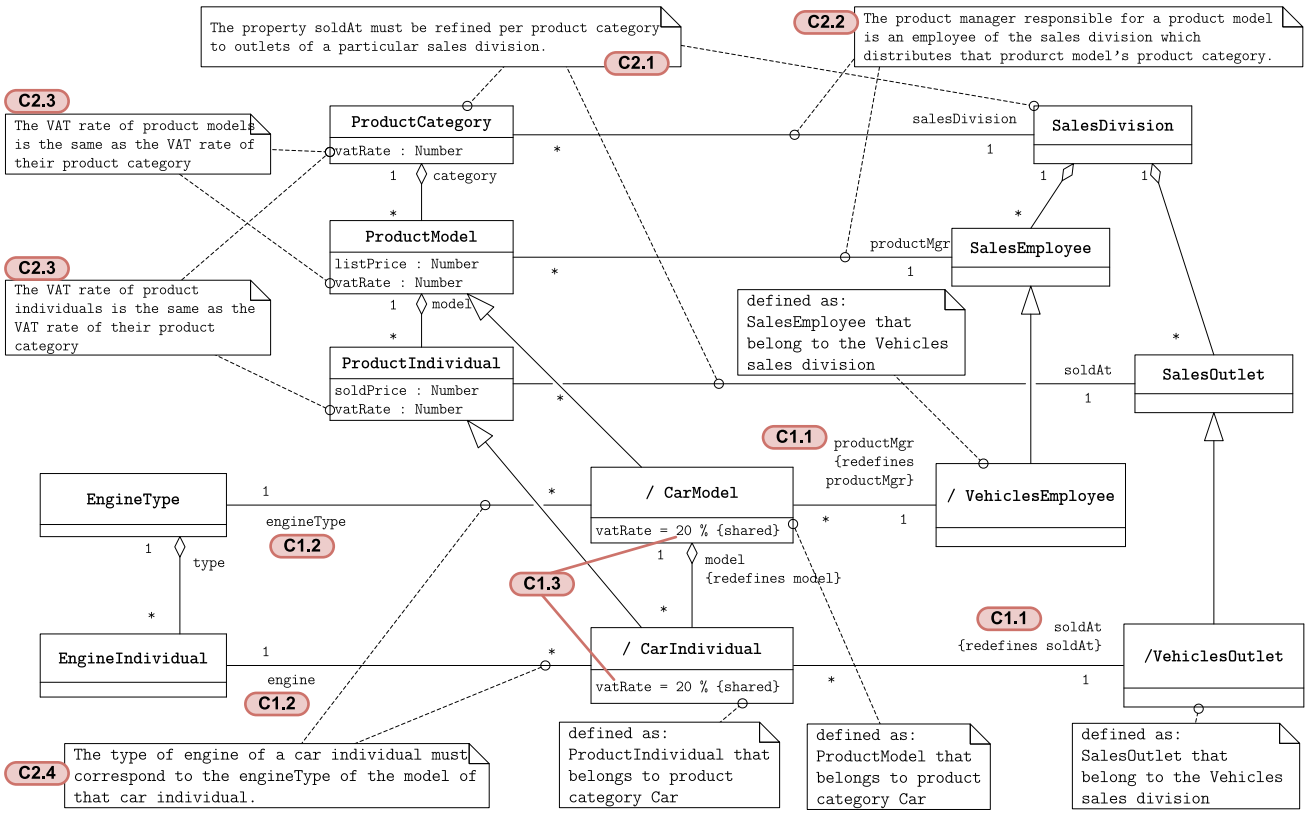


Fig. 3 Preliminary and partial solution to Modeling Challenge 1 (Specialization along the Hierarchy) and Challenge 2 (Regulating Specialization): specialization along the hierarchy is facilitated by derived classes like `CarModel` and `VehiclesOutlet`. As a work-around to regulated specialization, the to-be modeled regulations are expressed as custom constraints as far as possible. The labels C1.1–C2.3 refer to the to-be supported modeling tasks

values that are fixed by a class for every member of that class, are marked with `{shared}`.

The specializations regarding descendants of product category `Car` at levels Model and Individual are modeled with derived classes `CarModel` and `CarIndividual`. Derived class `CarModel` is modeled as a subclass of `ProductModel`, its members are the members of `ProductModel` which belong to product category `Car`. Derived class `CarIndividual` is modeled as a subclass of `ProductIndividual`, its members are the members of `ProductIndividual` which belong to product category `Car` (indirectly via a member of `CarModel`). Both, `CarModel` and `CarIndividual`, define `20%` as shared value for their `vatRate` property.

Derived classes `VehiclesEmployee` and `VehiclesOutlet` represent the descendants of sales division `Vehicles` at levels Employee and Outlet, respectively. Class `CarModel` refines the range of property `productMgr` to `VehiclesEmployee`. Class `CarIndividual` refines the range of property `soldAt` to `VehiclesOutlet`.

Assessment of preliminary solution We will now discuss the quality of the preliminary modeling solution with regard to the desiderata (see Sect. 3.1). The preliminary solution is *non-disruptive* (+) since the original conceptual model remains intact. *Conceptual clarity* (+) is reached by modeling properties that characterize descendants at a particular level with a derived class. *Compactness* (−) is weak since the derived classes introduced with domain object `Car` are not tied together with the domain object and only upon inspection of the classes' derivation rules it becomes clear that they belong to domain object `Car`. With regard to *factoring out commonalities* (∼), the modeling information that `Car` is under `Product` is manually duplicated as specialization between `CarModel` and `ProductModel` and `CarIndividual` and `ProductIndividual`. The solution does not *avoid custom constraints* (−) since the derivation rules for derived classes are encoded manually.

3.3 Regulating specialization along the hierarchy

The second challenge is to let the modeler govern specialization at a more generic level. Schema refinements can be modeled separately for each sub-hierarchy as above, but with domain object hierarchies in enterprise information systems

it comes natural that these refinements often follow common constraints and rules for every sub-hierarchy. We will, first, describe the modeling tasks and exemplify them along the running example. For the preliminary and only partial solution we will make heavy use of custom constraints on top of the UML class diagram (see Fig. 3) and we will assess the preliminary solution with regard to the desiderata (see Sect. 3.1).

The to-be supported modeling tasks are:

C2.1 The modeler should be able to specify that certain range refinements for property p of domain objects at level k must be made with every domain object at some level k' (where k is under k'), e.g.:

- A product individual cannot be sold at an arbitrary sales outlet but only at sales outlets that belong to a particular sales division. Such a refinement (i.e., restricting the range of property `soldAt`) must be made with every product category.

C2.2 The modeler should be able to 'automate' the range refinement of property p for objects at level k based on the value of a property p' of domain objects at level k' (where k is under k').

- The product manager responsible for a product model must be from the sales division which distributes the product category the product model belongs to.

C2.3 The modeler should be able to specify that the value of property p of domain objects at level k is propagated as shared value of a property p' to descendant objects at level k', e.g.,

- The VAT rate of a product model is the same as the VAT rate of the product category the product model belongs to.
- The VAT rate of a product individual, likewise, is the same as the VAT rate of the product category the product individual indirectly belongs to.

C2.4 The modeler should be able to specify such regulations also only for sub-hierarchies, e.g.,

- The type of an individual car's engine must correspond to the engine type associated with that car's car model.

Partial preliminary solution. In our preliminary solution (see Fig. 3) we found no way to model the obligation to refine a property to a certain level (see C2.1). All other modeling tasks could only be solved by custom constraints (expressed in natural language) attached to the properties.

Assessment of preliminary solution. The major shortcoming is obviously that the solution does not at all *avoid custom constraints* (−). The solution is non-disruptive (+). It comes, arguably, with conceptual clarity (+) because custom constraints are attached to the affected model elements. The solution does not care about *compactness* (−). There are no shortcomings with regard to *factoring out commonalities* (+) as there is no need to duplicate model information.

3.4 Information aggregation along the hierarchy

The third challenge is to model the aggregated information that should be provided with domain objects about its descendant objects at various hierarchy levels. We will first describe which aggregated information should be provided with the domain objects in the Product hierarchy. As a preliminary solution, we will model the aggregated properties in UML as derived properties. We will then assess this preliminary solution based on the desiderata (see Sect. 3.1).

The to-be supported modeling tasks are:

C3.1 The modeler should be able to specify an aggregated property p that provides aggregated information about property p' of domain objects at some level k and should provide values for aggregated property p with domain objects above level k.

- The sold price of product individuals should be aggregated to average sold price. This aggregated property should be available with every domain object in the Product hierarchy that is above the individual level.
- The list price of product models should be aggregated to average list price. This aggregated property should be available with every domain object in the Product hierarchy that is above the model level.

C3.2 It should be possible to define multi-step aggregations, i.e., aggregated properties that are calculated from aggregated properties, e.g.,

- The average sold price per product model should be further aggregated to a minimum average sold price per model and this aggregated property should be available with every domain object in the Product hierarchy that is above the model level.

C3.3 Aggregated information should be provided also for the domain object hierarchy as a whole.

- Values for the three aggregated properties should be available for the Product hierarchy as a whole.

Preliminary solution. In our preliminary solution (see Fig. 4), we introduce a singleton class `ProductHierarchy` the instance of which represents the overall product hierarchy. We model 'average sold price' multiple times as derived property `avgSoldPrice` with classes `ProductModel`,

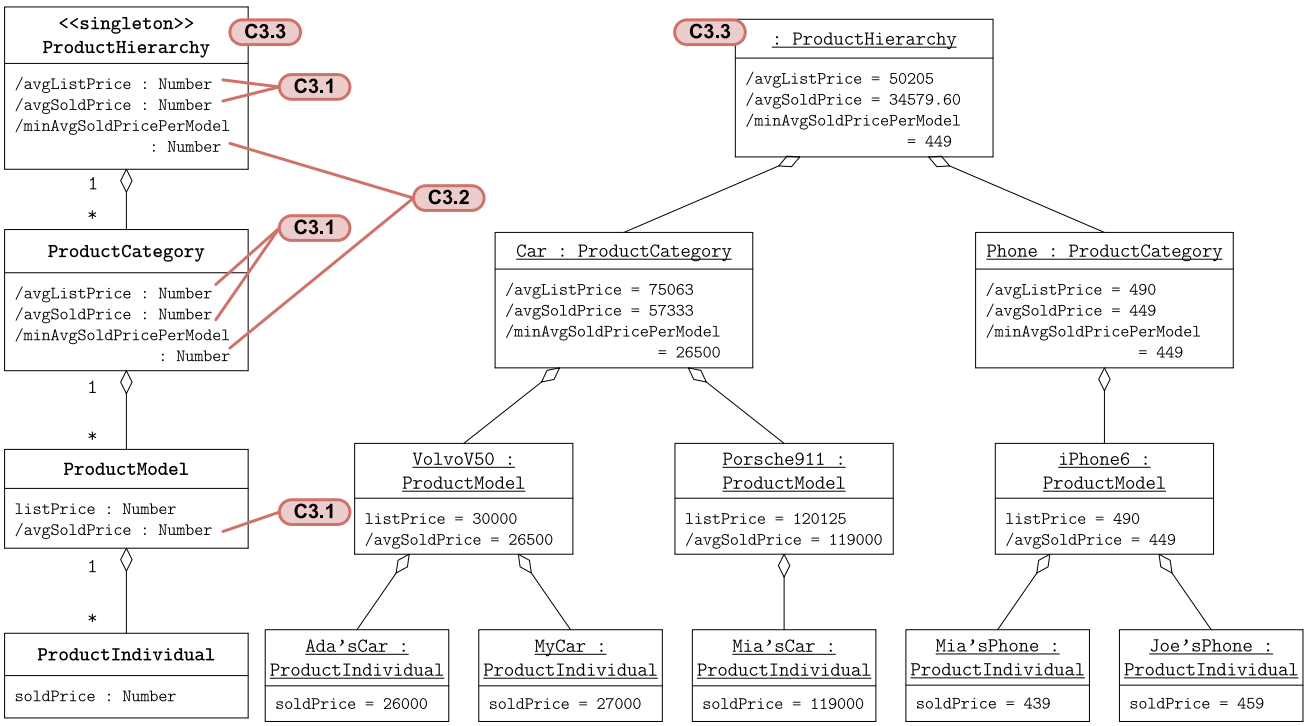


Fig. 4 Preliminary solution to Modeling Challenge 3 (Information aggregation along the hierarchy): aggregated properties are modeled as derived properties, the root of the hierarchy is represented by a singleton class

`ProductCategory`, and `ProductHierarchy`. Likewise, we model 'average list price' and 'minimum average sold price per model' multiple times as derived properties `avgListPrice` and `minAvgSoldPricePerModel` with classes `ProductCategory` and `ProductHierarchy`. The result of the information aggregation along the hierarchy (together with the asserted values for sold price and list price) is shown in the object diagram of Fig. 4.

Assessment of preliminary solution. The solution does not *factor out commonalities* (−) but models aggregated properties like `avgSoldPrice` multiple times with the different classes. It does not come with *conceptual clarity* (−) since the model does not get across with aggregated information from which set of objects it is calculated (one would have to inspect the derivation rule which we have not modeled). The solution is *non-disruptive* (+) since the solution only adds to the existing model and comes with a high degree of *Compactness* (+), and there is no need for custom constraints (+).

4 Towards multi-level modeling of domain object hierarchies

In this section we discuss how to model aspects of the sample domain object hierarchy using existing multi-level modeling techniques, namely Carvalho et al.'s Multi-Level Theory (MLT) [8] and our previous work on Dual Deep Modeling (DDM) [30]. These two MLM approaches serve as starting point for our further considerations.

4.1 MLT-based modeling of domain object hierarchies

Domain object hierarchies modeled as part-whole hierarchies come with a uniform treatment of physical objects like `MyCar` and more class-like nonphysical objects such as product model `VolvoV50`. Instead of regarding the latter as classes we treated them as nonphysical objects that have a one-to-one correspondence with a class, e.g., `VolvoV50Individual`. With MLT one typically takes another approach and represents by `VolvoV50` both the product model `VolvoV50` and the class `VolvoV50Individual` of physical cars of product model Volvo V50. In this subsection let us model the Product hierarchy accordingly (see Fig. 5 for an UML-like representation).

Domain objects at the lowest level (e.g., product individual `MyCar`) are simple objects. Domain objects at higher levels, such as product model `VolvoV50` and product category `Car`, are modeled as first-order classes. Non-bottom levels of the product hierarchy are modeled as 2nd-order classes `ProductModel` and `ProductCategory`. The bottom level of the product hierarchy (represented in Fig. 2 by class `ProductIndividual`) is represented as first-order class

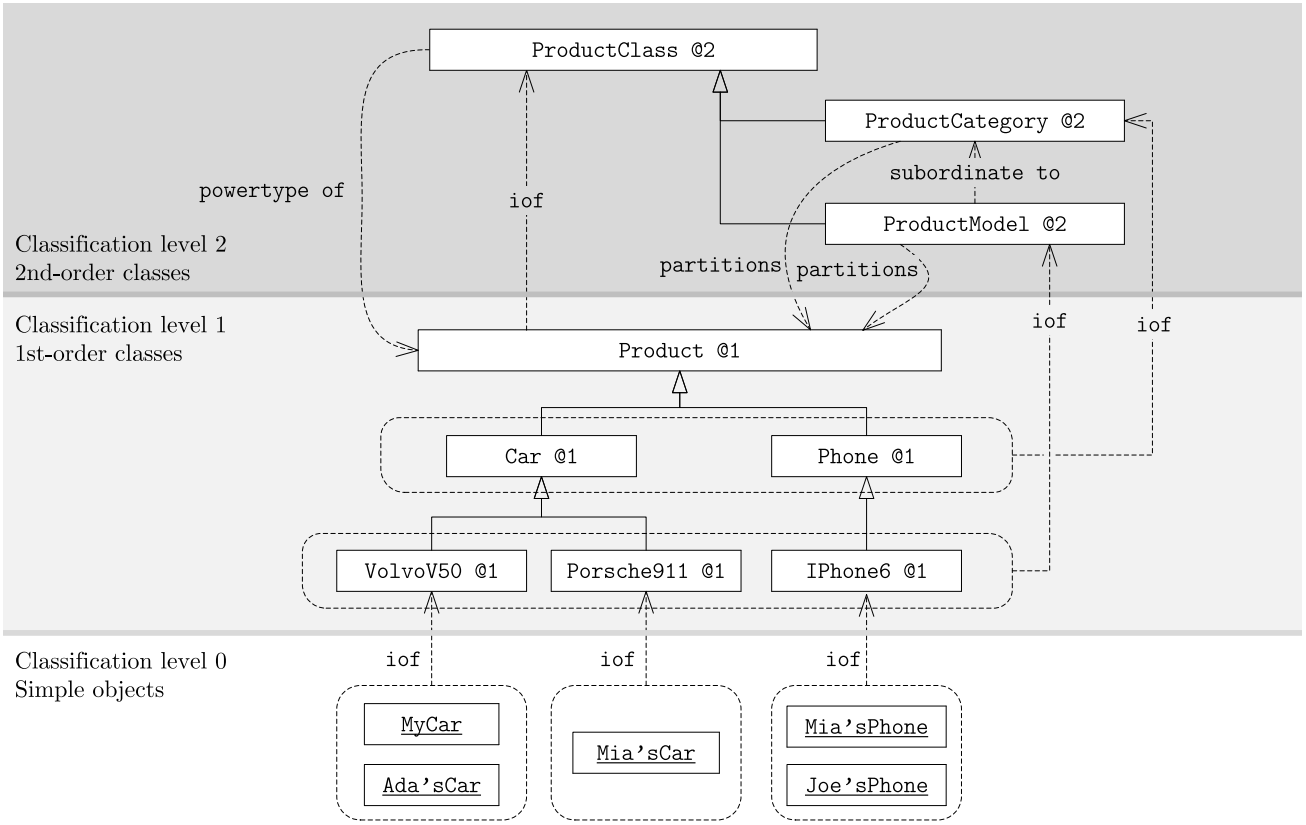


Fig. 5 A product hierarchy modeled in UML style extended with level-crossing dependencies from Carvalho et al.'s multi-level theory [8]. Multiple objects/classes having a dependency with the same target are collected into a group with the dependency only shown once for the group, e.g., `MyCar` and `Ada'sCar` are both instances of `VolvoV50`

`Product`—this adjustment in naming is to make the models conformant to the naming conventions applied in the examples found in the literature on MLT.

The hierarchy of domain objects that are not at the bottom level is represented as a subclass hierarchy. For example, `VolvoV50` is a subclass of `Car`, which is a subclass of `Product`, which makes `VolvoV50` also a (indirect) subclass of `Product`. As an instance of class `VolvoV50`, `MyCar` is also a member of class `Car` and of class `Product`.

Higher-order classes are also organized in subclass hierarchies. Second-order classes `ProductCategory` and `ProductModel` are sub-classes of `ProductClass` with `Car` being an instance of `ProductCategory` and, hence, a member of `ProductClass`.

Now let us use MLT to specify in more detail the kinds of second-order and higher-order classes in our multi-level model.

The first kind of metaclass we are interested in is the *power type* as introduced by Cardelli [7] and incorporated into MLT. The power type of a class (which is the power type's base class) has all subclasses of that class, including that class, as members. For example, in Fig. 5, metaclass `ProductClass` is the power type of class `Product`. All sub-classes of `Product`, including `Product` are members of `ProductClass`.

The second kind of metaclass relevant to the multi-level modeling of domain object hierarchies is the kind of power type introduced by Odell [31]. We limit the use of Odell's power types to what in MLT [8] is referred to as disjoint and complete categorization, or *partitioning*. Note, the restriction to disjoint and complete categorization is because we assume every domain object at a lower level in a domain object hierarchy to belong to exactly one domain object at the next higher level. We refer to this kind of metaclass not as power type but to as *partitioning metaclass*, or simply as *metaclass*, since in our approach all metaclasses, apart from Cardelli's power types are partitioning metaclasses.

A partitioning metaclass always comes together with a partitioned class, also referred to as base class. Every member of the partitioning metaclass is a (direct or indirect) subclass of its base class and every member of the partitioned class is member of exactly one member of the partitioning metaclass. For example, the metaclass `ProductModel` partitions class `Product`. Its members, such as `VolvoV50`, are sub-classes

of Product. Every member of partitioned class Product is member of exactly one member of ProductModel, for example, MyCar is member of VolvoV50.

Also relevant to the multi-level modeling of domain object hierarchies is the notion of subordination between metaclasses, as introduced with MLT [8]. If a metaclass is subordinate to another metaclass, then every member of the former is subclass of some member of the latter. For example, the metaclass ProductModel is subordinate to metaclass ProductCategory. Every member of ProductModel is sub-class of one member of ProductCategory, for example, VolvoV50 is subclass of Car.

Discussion. The modeling of higher-level domain objects, such as VolvoV50 and Car, as classes seems very natural. The same is true for the modeling of classes, such as ProductModel and ProductCategory, as metaclasses. One may argue, that the change in semantics from the original model better reflects the ontological nature of things and avoids accidental complexity [4]. Yet, other authors [11] argue quite the opposite, namely that concepts like VolvoV50 are, first and foremost, objects.

What we aim for with the DDO pattern and approach is broader applicability of multi-level modeling without sacrificing conceptual clarity. What motivates our research in this direction is the observation that many domain object hierarchies, like the Sales organisation hierarchy in our running example, are of a very different nature, cannot be adequately modeled using the multi-level modeling style exemplified in this section, yet also induce subclasses and metaclasses. The following example also illustrates this observation: A domain object hierarchy of geographical entities, e.g., Innsbruck is a city in Tyrol, and Tyrol is a state in Austria, induce multi-level models, e.g., Innsbruck is a member of class 'Tyrolean city' which in turn is a subclass of 'Austrian city' as well as a member of metaclass 'Austrian city class by state'.

Our intuition here is that all these different domain object hierarchies are different kinds of part-whole hierarchies and that the modeling of part-whole comes prior to the modeling of subclasses and partitioning metaclasses. This is related to Partridge et al's work [32] which is driven by Kit Fine's theory of part [12]. Fine "takes a very liberal notion of part" [32] with subset and set-membership as kinds of part-whole. With sets considered as wholes, Fine argues for "taking both its members and its subsets to be parts". Also, our distinction between modeled domain objects and induced classes and metaclasses relates well to the works of Partridge et al. and Fine. They distinguish a basis domain, a constructor domain, and a constructed domain (cf. Table 1 in [32]). We consider domain objects and their arrangement in a domain object hierarchy as given, as 'basis domain', and the induced classes and metaclasses as 'constructed domain'.

Currently, we only consider domain object hierarchies where a domain object always belongs to exactly one domain object at the next higher level. That is why we are only interested in partitioning metaclasses and can avoid multiple inheritance. A partial lifting of these constraints, which will make relevant MLT's remaining cross-level relations (categorizes, disjointly categorizes, completely categorizes), is left to future work.

4.2 Higher-order partitioning metaclasses

What seems particular relevant to the multi-level modeling of domain object hierarchies are higher-order partitioning metaclasses. Higher levels of a domain object hierarchy not only partition the domain object hierarchy's bottom level but also intermediate levels. With intermediate levels already being represented as second-order classes, this gives rise to 3rd-order classes. This can be applied recursively: a domain object hierarchy's n-th level (from the bottom) can be represented as 2nd . . . nth-order classes, each partitioning a 1st . . . $(n-1)$th-order class.

For example, level Category of the domain object hierarchy not only partitions the set of *product individuals* but also the set of *product models*. With the latter represented by 2nd-order class ProductModel, we introduce a 3rd-order class ProductModelClassByCategory which partitions it (see Fig. 6) into sub-classes CarModel and PhoneModel.

Discussion. So far we have discussed the representation of domain object hierarchies with classes and meta-classes applying some of the constructs of MLT, namely *power type*, *partitioning*, and *subordination*. As already evident with these small examples the modeling of domain object hierarchies using these constructs is elaborate and results in well understandable but verbose models. A major drawback is that a single domain object (e.g., product category Car) and a single hierarchical relationship (e.g., Car belongs to Product) have to be represented by multiple classes and relationships, respectively, in the multi-level model.

This observation led us to the conviction that large parts of multi-level models with precise semantics need not be modeled manually but can be derived from abstraction hierarchies modeled as part-whole hierarchies that do not, as such, come with precise semantics.

One possible limitation of our view on domain object hierarchies is that we treat all hierarchical relationships in a domain object hierarchy the same and distinguish between subset and set membership only with respect to induced classes. This can make the application of the DDO approach unnecessarily heavyweight, since potentially irrelevant classes and metaclasses, as well as their subclass-of and instance-of relationships, are also made explicit.

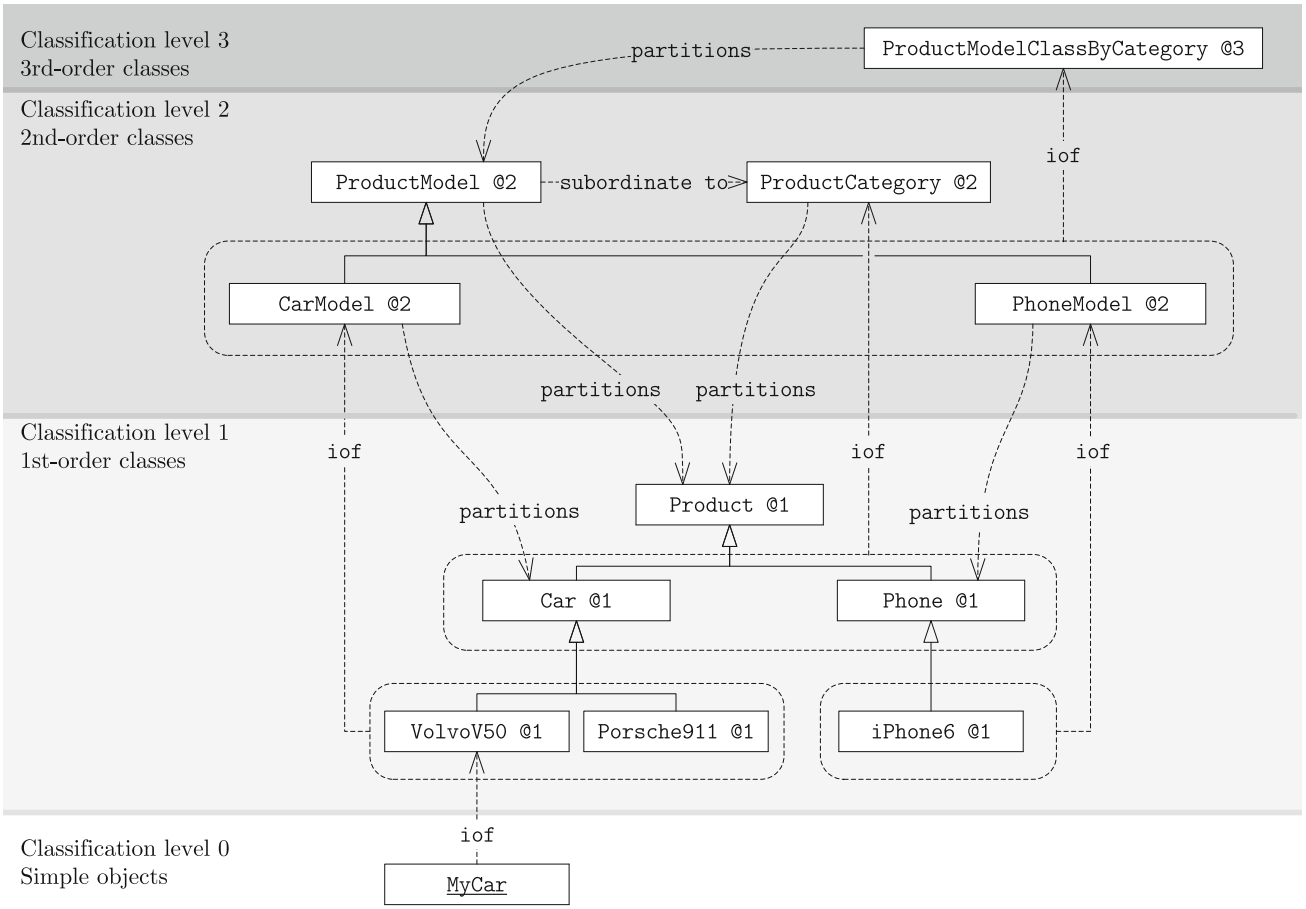


Fig. 6 A fragment from the product hierarchy with higher-order partitioning duplicating hierarchical relationships at different classification levels

4.3 Multi-level modeling with named levels in DDM

Now let us revisit our previous work on Dual Deep Modeling (DDM) [30]. DDM facilitates more compact multi-level models, yet lacks the conceptual clarity of MLT.

The core construct of the DDM approach is the DDM clabject. DDM clabjects are connected by properties with dual potencies, a source potency and a target potency. The source and target potencies of a property indicate the number of instantiation steps at the source and, respectively, target end of the property to reach an ultimate instance of the property. Properties in DDM can have inverse properties, can be arranged in specialization hierarchies, may be multi-valued and can be further characterized by multi-level cardinality constraints. Although DDM's powerful and flexible property mechanism will guide our future work to develop a fully-fledged property mechanism in DDO, it is clearly beyond the scope of this paper.

What we are interested in this paper is DDM's use of named levels which go back to our work on M-Objects [28].

In Fig. 7 (left part), a product hierarchy is represented by DDM clabject `Product` with levels Category, Model, and Individual, where Individual is under Model is under Category. Third-order clabject `Product` is instantiated by a second-order clabject `Car` at level Category which in turn is instantiated by a first-order clabject `VolvoV50` at level Model which in turn is instantiated by a clabject `MyCar` at level Individual.

What led us to the DDO approach was the realization that a DDM clabject actually represents a nested structure of local classes of ascending order (illustrated by Fig. 7), induced as follows.

Going back upwards the instantiation hierarchy we say, now more precisely, `MyCar` is an instance of `VolvoV50Individual`, the Individual class local to car model `VolvoV50`. `VolvoV50` is an instance of `CarModel`, the Model class local to product category `Car`; and `Car` is an instance of `ProductCategory`, the Category class local to root object `Product`.

Now, turning to the classification of induced classes along the hierarchy, we say `VolvoV50Individual` is an instance of `CarIndividualClassByModel`, the Individual metaclass local to `CarModel`. Class `CarModel`

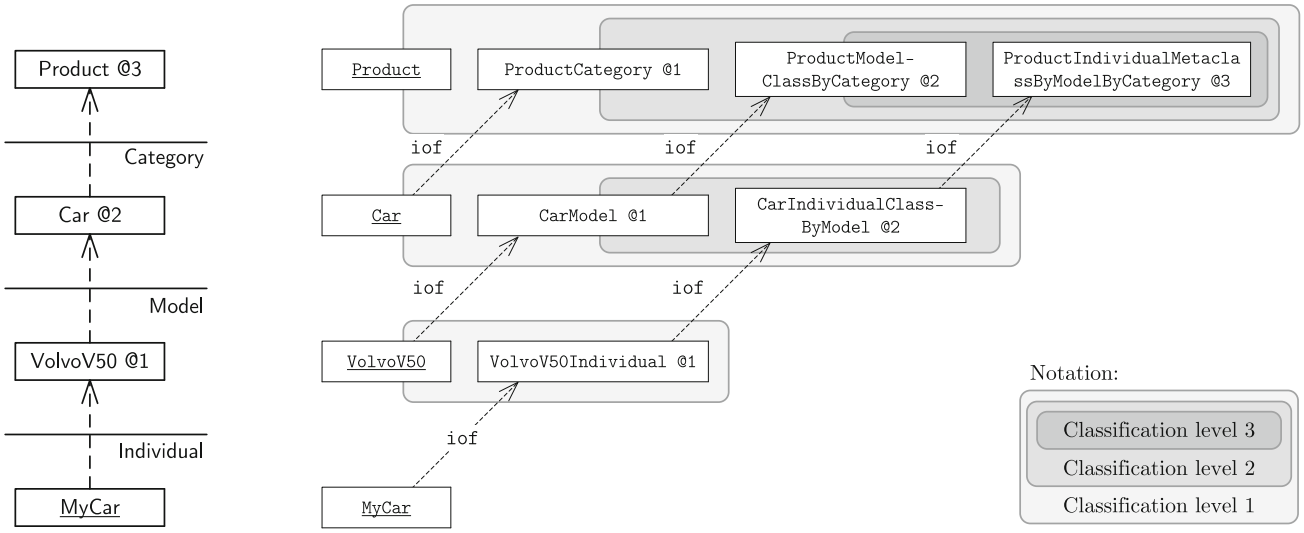


Fig. 7 Left: product hierarchy modeled with deep instantiation and named levels as promoted with the DDM approach [30]. Note, in DDM there is no distinction between domain object hierarchy levels and classification levels. Right: The product hierarchy modeled with local meta*-classes, the basic idea underlying Dual Deep Modeling [30] (not elaborated as such in [30]). Local meta*-classes get a compound name reflecting their nesting and are shown with their order or potency. Different shades of grey represent the different classification levels of the multi-level model. Nesting of rounded rectangles represents *local to* relationships, e.g., first-order class `CarModel` is local to object `Car`, and 2nd-order class `CarIndividualClassByModel` is local to class `CarModel` and is indirectly local to object `Car`

is an instance of `ProductModelClassByCategory`, which is the Model metaclass local to `ProductCategory`.

Finally, turning to the classification of induced metaclasses, we can say that metaclass `CarIndividualClassByModel` is an instance of `ProductIndividualMetaclassByModelByCategory`, which is the Individual meta-meta class local to metaclass `ProductModelClassByCategory`.

Summarizing, the instantiation of a DDM clabject of order n by a clabject of order $n-1$ can be regarded as n closely coupled instantiations at different levels of classification.

Discussion. The clear picture of the meaning of DDM clabjects we have painted here, fits perfectly with our intuition behind DDM. However, DDM's highly flexible and powerful property mechanism including support for skipping levels when instantiating properties may obscure this clear picture.

In order to more clearly convey the meaning of induced multi-level models, we will, in this paper, employ a highly simplified and more strict property mechanism, only considering single-valued, uni-directional properties without support for skipping levels.

Another shortcoming of DDM is the lack of support to address superclasses induced along the hierarchy, for example, generalizing `VolvoV50Individual` to superclass `CarIndividual`, which in turn generalizes to superclass `ProductIndividual` . This shortcoming is mitigated by DDM's powerful and flexible property mechanism, not discussed here in detail, which allows to use higher-order classes as superclasses, for example metaclass `CarIndividualClassByModel` can be used in many regards like it was a superclass `CarIndividual`, what further diminishes conceptual clarity.

As a major addition in comparison to DDM, the DDO approach will make all the induced superclasses (at various classification levels) directly addressable by the modeler. The explicit representation of induced superclasses adds to the modeling power and improves conceptual clarity.

Furthermore, DDM's reliance on numeric potencies limits its use to the modeling of domain object hierarchies the levels of which are in a total order. By getting rid of numeric potencies and instead fully relying on level names, the DDO approach can also support the multi-level modeling of domain object hierarchies the levels of which are only in a partial order, such as the Sales organisation hierarchy. The use of level names would also facilitate the introduction of intermediate levels in sub-hierarchies (not discussed in this paper) as featured by the M-Objects [28] approach.

5 From part-whole to multi-level hierarchy

In this section we first introduce the deep domain object (DDO) pattern for representing classes and metaclasses induced by a domain object hierarchy. The *DDO approach* to multi-level modeling has at its core the automatic and recursive application of the DDO pattern, which we will formalize in this section by a set of deductive rules. When working with the DDO approach, the modeler does not have to care about

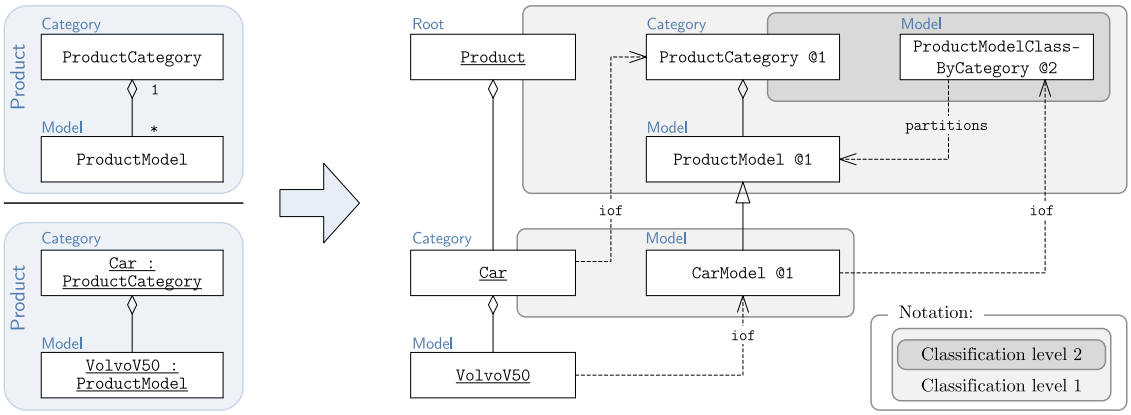


Fig. 8 The DDO pattern: a domain object hierarchy with two levels modeled as part-whole hierarchy (left) induces a multi-level model (right) comprising local classes, a metaclass, and their subclass-of, direct-instance-of (iof), and partitions dependencies. The rounded rectangles with grey background show the nesting structure of local classes, e.g., class `ProductCategory` is local to domain object `Product`, and of metaclasses, e.g, metaclass `ProductModelClassByCategory` is local to class `ProductCategory`. The shade of grey indicates the nesting depth which corresponds to the order of the classes, also referred to as classification level

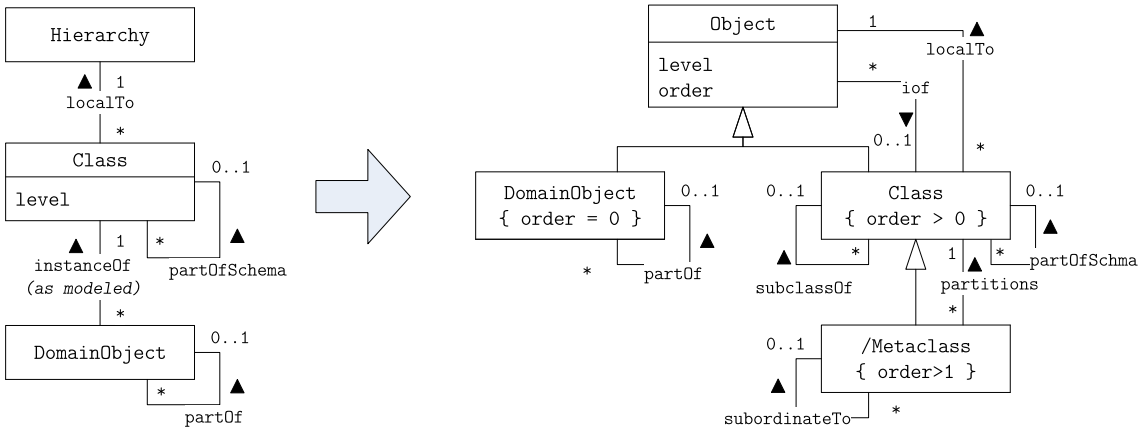


Fig. 9 Metamodel of domain object hierarchy modeled as part-whole hierarchy (left). Metamodel of domain object hierarchy as induced multi-level model (right). Note that the given associations and multiplicities model direct relationships/dependencies and not their transitive or reflexive-transitive closures

induced subclasses and metaclasses, yet can use them when needed.

The approach is illustrated along our running example. Figure 8 exemplifies the DDO pattern with a small fragment of the product hierarchy. Figure 9 shows the metamodel of domain object hierarchies modeled as part-whole hierarchies and of domain object hierarchies with induced classes and metaclasses. Figures 10 and 11 exemplify the application of the pattern for a fragment of the Product hierarchy (a domain object hierarchy with three levels in a total order) and of the Sales organisation hierarchy (a domain object hierarchy where the levels form a tree and not a path).

For the deductive rules we apply minimal Herbrand model semantics and the closed-world assumption for negation as in Datalog$^\neg$ (cf. [19]) plus existentially quantified rule heads for creating new model elements. Rules are being invoked alternating and recursively until a fixpoint is reached and a minimal Herbrand model found (in our case representing the induced multi-level model).

In this section we focus on deriving the multi-level structures and multi-level dependencies induced by part-whole hierarchies. With the DDO approach we further assume that every (induced) class has a power type which has the class and all its direct and indirect subclasses as members. Since this is not specific to the DDO approach we refrain from providing formalization of power types and do not show them in the multi-level models except when used for modeling aggregated properties. A complete realization of the approach (including power types) and of the running exam-

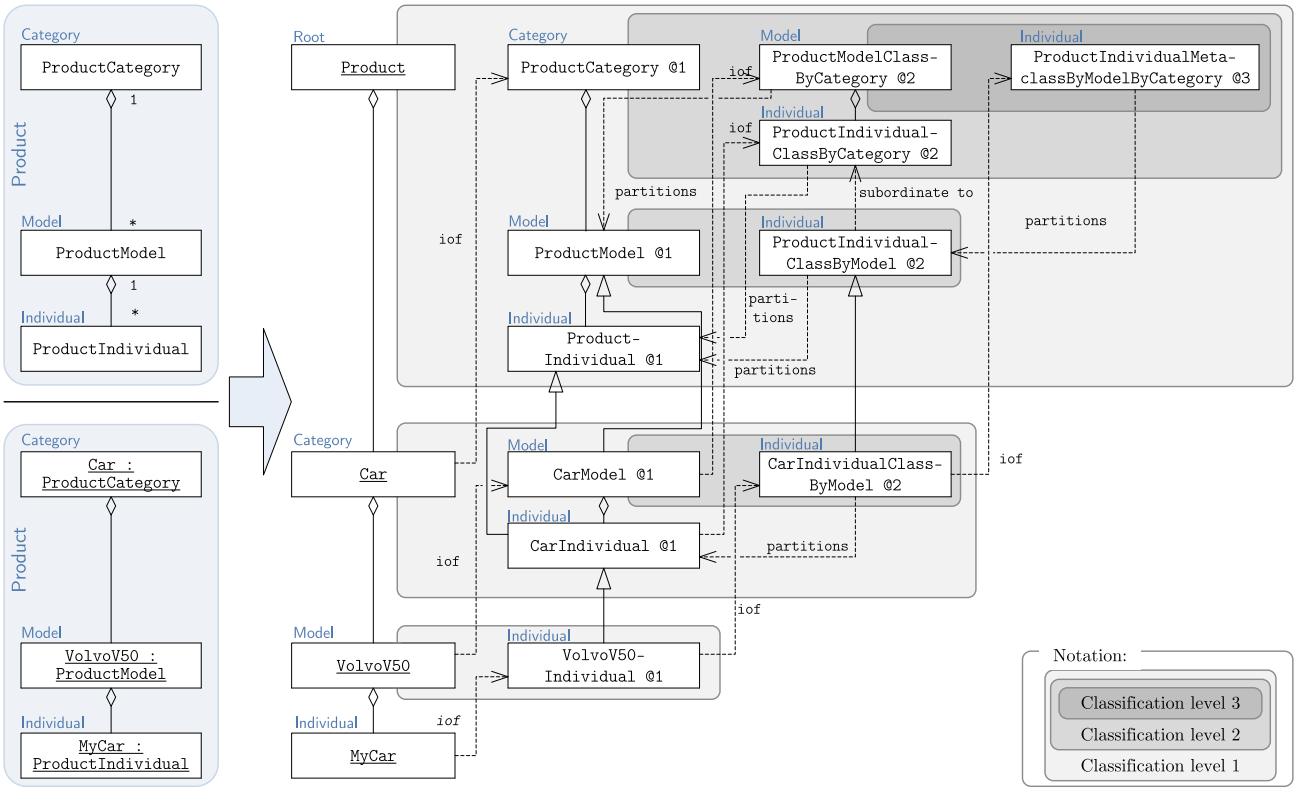


Fig. 10 The DDO pattern applied on the Product hierarchy with its three levels which are arranged in a total order. In comparison to Fig. 8, the additional Individual level gives rise to additional classes, metaclasses, a meta-metaclass, and a subordinate-to dependency between metaclasses. Note, the `partitions` and `subordinateTo` dependencies are not an essential part of the induced multi-level model yet clarify its semantics in terms of MLT [8]

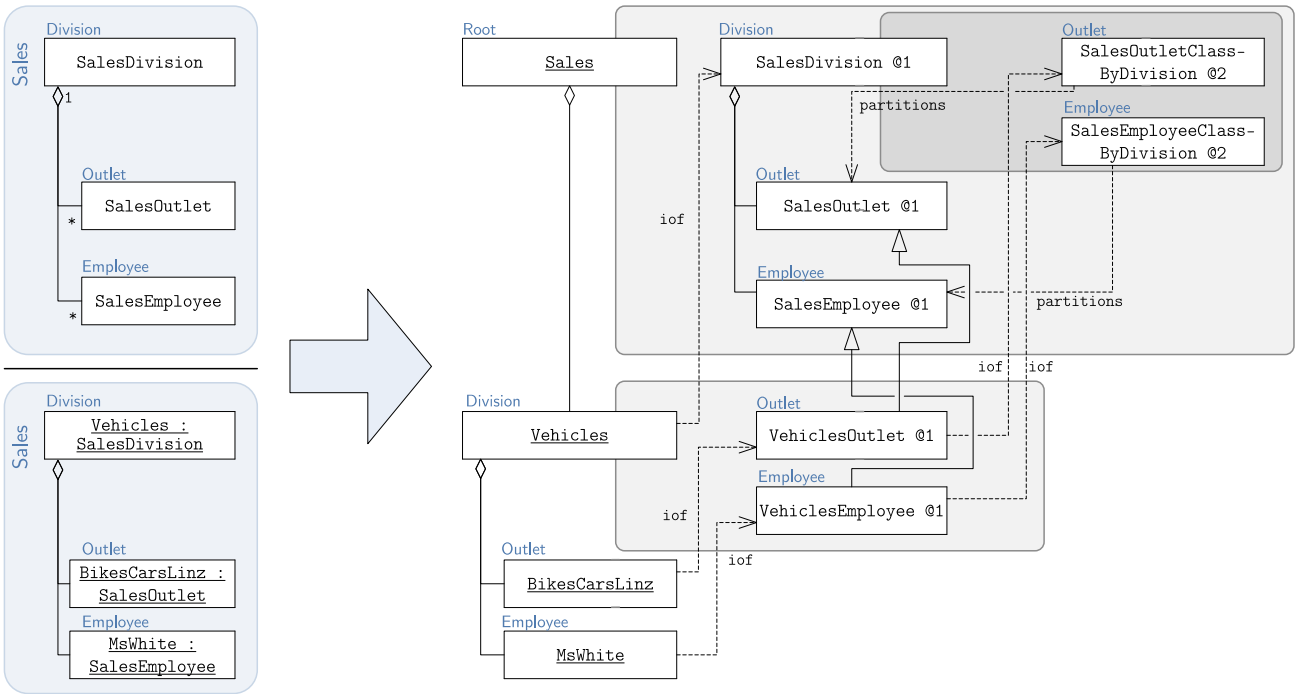


Fig. 11 The DDO pattern applied on the Sales organisation hierarchy with its tree-like schema (levels Outlet and Employee are directly under level Division)

ple is provided in F-Logic (see Appendix) which also serves as proof-of-concept implementation (see also Sect. 7).

5.1 The DDO pattern

Let us first introduce in simple terms the Deep Domain Object pattern (DDO pattern) for representing domain object hierarchies and their induced classes and metaclasses. The DDO pattern is a multi-level modeling pattern that builds on 'promotion with base classes' [24], a solution to the type-object pattern, and makes use of metaclasses and cross-level dependencies in accordance with the multi-level theory (MLT) [8].

A key principle of the DDO pattern is that induced classes are modeled local to the domain object by which it is induced. In this way, the DDO pattern fulfills the *compactness* desideratum of Sect. 3.1.

We will introduce the DDO pattern for a domain object hierarchy fragment with only two levels, each with only one domain object. At the schema level the hierarchy, e.g., Product, is modeled by a component class, e.g., `ProductModel`, associated via aggregation relationship to a composite class, e.g., `ProductCategory`, and at the instance level by a component object, e.g., `VolvoV50`, linked via aggregation to a composite object, e.g., `Car`. The result of applying the pattern is shown in the right part of Fig. 8. The DDO pattern consists of:

1. *Root object:* a domain object hierarchy as a whole is represented by a root domain object. In this way, the domain object hierarchy, as well as its sub-hierarchies, are each uniformly represented by a domain object. The classes representing the schema of the hierarchy are modeled *local to* the hierarchy's root object.
   For example, the Product hierarchy is represented by root domain object `Product`. Composite object `Car` is modeled as part of `Product`. Classes `ProductCategory` and `ProductModel` are modeled as local to `Product`.
2. *Induced subclasses:* with every composite object, i.e., every instance of the composite class, the modeler introduces, *local to* the composite object, a subclass of the component class. The induced subclass can then be used for specializing the schema.
   For example, with the product category `Car` the modeler introduces the class `CarModel` as the subclass of `ProductModel` local to `Car`. This is akin to 'promotion with base classes' [24], a solution to the type-object pattern, but additionally with a cross-level *local-to* dependency.
3. *Induced metaclasses:* the modeler further introduces, *local to* the composite class, a metaclass which partitions (as in MLT [8]) the component class. When instantiating the composite class, also the partitioning metaclass gets instantiated, namely by the subclass of the component class. The induced metaclass can be used by the modeler to regulate the specialization of the component class with regard to composite objects.
   For example, when instantiating `ProductCategory` by `Car`, metaclass `ProductModelClassBy Category`, which is local to `ProductCategory` and partitions `ProductModel`, gets instantiated by `CarModel`.

Applying the DDO pattern by hand is cumbersome and not feasible for all but very small domain object hierarchies. In the following we will formalize the pattern's automatic and recursive application.

5.2 Preparing the domain object hierarchy

The first step is to prepare the part-whole hierarchies for uniform treatment of (the roots of) hierarchies and sub-hierarchies as domain objects. Every hierarchy is treated as a domain object that represents the whole hierarchy and acts as root of the hierarchy [rule (A1)]. Then all domain objects at a top-most level of the hierarchy are assigned as parts of that root object (A2). Domain objects are annotated with the hierarchy level represented by their class (A3) and are assigned to classification level 0, i.e., their order is 0 (A4). For example (see Fig. 10), domain object hierarchy Product becomes domain object `Product` at level Root representing the whole product hierarchy with product category `Car` as part. Domain objects like `Car` are annotated with their level, e.g., Category, and are associated with classification level 0.

Note, we use predicate instanceOf for the asserted instance-of relation, e.g., `VolvoV50: ProductModel` is represented as instanceOf(`VolvoV50`,`ProductModel`), and use predicate iof for the direct instance of relation which takes into account induced sub-classes, e.g., iof(`VolvoV50`,`CarModel`).

$$\forall h : \mathsf{Hierarchy}(h) \Rightarrow \mathsf{DomainObject}(h) \wedge \mathsf{level}(h, \mathsf{Root}). \tag{A1}$$

$$\forall o, h, c : \mathsf{instanceOf}(o, c) \wedge \mathsf{localTo}(c, h) \wedge (\nexists c' : \mathsf{partOfSchema}(c, c')) \Rightarrow \mathsf{partOf}(o, h) \tag{A2}$$

$$\forall o, c, k : \mathsf{instanceOf}(o, c) \wedge \mathsf{level}(c, k) \Rightarrow \mathsf{level}(o, k) \tag{A3}$$

$$\forall o : \mathsf{DomainObject}(o) \Rightarrow \mathsf{order}(o, 0) \tag{A4}$$

5.3 Deriving multi-level instantiation hierarchies

In the following we define the rules that derive multi-level instantiation hierarchies of local classes.

Given is a domain object o which is part of domain object o' and modeled as instance of class c which represents level

k. Then domain object o is direct instance of (iof) the class c' local to o' which also represents level k. For example, product category `Car` is part of domain object `Product` and modeled as instance of `ProductCategory` which represents level Category. Now, since `ProductCategory` is local to `Product`, it follows that `Car` is a direct instance of `ProductCateogry`.

$$\begin{aligned}&\forall o, o', c, c', k : \mathsf{partOf}(o, o') \wedge \mathsf{instanceOf}(o, c)\\ &\quad \wedge \mathsf{level}(c, k) \wedge \mathsf{localTo}(c', o') \wedge \mathsf{level}(c', k)\\ &\quad \Rightarrow \mathsf{iof}(o, c')\end{aligned} \tag{A5}$$

Given is an object o which is a direct instance of (iof) class c which has class c' representing level k as direct or indirect part ($\mathsf{partOfSchema}^+$ represents the transitive closure of partOfSchema). Then a new class c'' local to o and representing level k comes into existence. For example, `Car` is a direct instance of `ProductCateogry` which has classes `ProductModel`, representing level Model, and `ProductIndividual`, representing level Individual, as direct or indirect parts. This induces local class `CarModel`, representing level Model for the subhierarchy rooted in `Car`, and class `CarIndividual`, representing level Individual for the subhierarchy rooted in `Car`. For the mechanism to generate names of local classes and metaclasses see the F-Logic implementation in the Appendix.

$$\begin{aligned}&\forall o, c, c', k : \mathsf{iof}(o, c) \wedge \mathsf{partOfSchema}^+(c', c) \wedge \mathsf{level}(c', k)\\ &\quad \Rightarrow \exists^{\mathsf{new}} c'' : \mathsf{localTo}(c'', o) \wedge \mathsf{level}(c'', k)\end{aligned} \tag{A6}$$

By alternating recursive invocation, rules (A6) and (A5) produce first-order classes local to domain objects further down the hierarchy, such as `VolvoV50Individual` local to domain object `VolvoV50`.

Next we define the rules to derive local metaclasses. Consider a class c that has class c' representing level k as direct or indirect part and the order of c is lower than a constant maxOrder (which can be set by the modeler). Then a new metaclass c'', local to class c and representing level k, comes into existence. For example, the class `CarModel` has `CarIndividual` as part which represents level Individual. This induces metaclass `CarIndividualClassByModel`, which represents level Individual local to class `CarModel`. Further metaclasses induced in this way are `ProductModelClassByCategory`, `ProductIndividualClassByCategory`, `ProductIndividualClassByModel`, and finally 3rd order class `ProductIndividualMetaclassByModelByCategory`.

$$\begin{aligned}&\forall c, c', k, i : \mathsf{partOfSchema}^+(c', c) \wedge \mathsf{level}(c', k)\\ &\quad \wedge \mathsf{order}(c, i) \wedge i < \mathsf{maxOrder}\\ &\quad \Rightarrow \exists^{\mathsf{new}} c'' : \mathsf{localTo}(c'', c) \wedge \mathsf{level}(c'', k)\end{aligned} \tag{A7}$$

With local metaclasses derived we also need to derive the instance of (iof) relationships between classes and metaclasses. Consider a metaclass m representing level k local to class c, and a class c' also representing level k local to o which is an instance of c. Now we can derive that c' is an instance of m. Consider, for example, the metaclass `ProductIndividualClassByCategory` which is local to class `ProductCategory` which has `Car` as an instance, and class `CarIndividual` which is local to `Car`. We can derive that `CarIndividual` is a direct instance of `ProductIndividualClassByCategory`.

$$\begin{aligned}&\forall m, c, k, o, c' : \mathsf{localTo}(m, c) \wedge \mathsf{level}(m, k)\\ &\quad \wedge \mathsf{iof}(o, c) \wedge \mathsf{localTo}(c', o) \wedge \mathsf{level}(c', k)\\ &\quad \Rightarrow \mathsf{iof}(c', m)\end{aligned} \tag{A8}$$

The order of a local class is the increment of the order of the object the class is local to.

$$\forall c, o, i : \mathsf{localTo}(c, o) \wedge \mathsf{order}(o, i) \Rightarrow \mathsf{order}(c, i+1) \tag{A9}$$

The *part-whole* hierarchy modeled under a class is propagated to metaclasses local to that class. Consider class c'' which is above classes c and c' in the part-whole hierarchy. When class c is directly under class c' then this direct part-whole relationship is propagated to metaclasses local to c''. According to (A10), when class c which represents level k is a direct part of c' which represents level k' and both are direct or indirect parts of c'' and metaclasses m, representing level k, and m', representing level k', are local to c'', then m is part of m'. For example, `ProductIndividual` is directly under `ProductModel` and both are under `ProductCategory`, thus metaclass `ProductIndividualClassByCategory` has a direct part-whole relationship with `ProductModelClassByCategory` at classification level 2.

$$\begin{aligned}&\forall c, c', c'', k, k', m, m' : \mathsf{partOfSchema}^+(c, c'')\\ &\quad \wedge \mathsf{partOfSchema}^+(c', c'') \wedge \mathsf{partOfSchema}(c, c')\\ &\quad \wedge \mathsf{level}(c, k) \wedge \mathsf{level}(c', k')\\ &\quad \wedge \mathsf{localTo}(m, c'') \wedge \mathsf{level}(m, k)\\ &\quad \wedge \mathsf{localTo}(m', c'') \wedge \mathsf{level}(m', k')\\ &\quad \Rightarrow \mathsf{partOfSchema}(m, m').\end{aligned} \tag{A10}$$

The *part-whole* hierarchy modeled with metaclasses is propagated to their instances. According to (A11), when classes c and c' are local to the same object o, and c is instance

of metaclass m, and c' is instance of metaclass m', and m is part of m', then c is derived to be part of c'.

$$\forall o, c, c', m, m' : \mathsf{localTo}(c, o) \wedge \mathsf{localTo}(c', o) \wedge \mathsf{iof}(c, m) \wedge \mathsf{iof}(c', m') \wedge \mathsf{partOfSchema}(m, m') \Rightarrow \mathsf{partOfSchema}(c, c') \quad \text{(A11)}$$

5.4 Deriving generalization hierarchies

With the multi-level instantiation hierarchies of local classes in place we can now derive subclass-of relationships at all classification levels. We only show how to derive direct subclassOf relationships. From there one can easily derive the reflexive-transitive closure which is often used instead of only considering direct subclass-of relationships.

A first-order class local to a domain object is subclass of the class that is local to the domain object's parent and represents the same level. According to (A12) when domain object o is part of domain object o' and o has a local class c representing level k and o' has a local class c' representing level k, then c is a subclass of c'. For example, `VolvoV50` is part of product category `Car`. Its local class `VolvoV50Individual` which represents level Individual in the subhierarchy rooted in `VolvoV50` is a subclass of `CarIndividual` which represents level Individual for the subhierarchy rooted in `Car`.

$$\forall o, o', c, c', k : \mathsf{partOf}(o, o') \wedge \mathsf{localTo}(c, o) \wedge \mathsf{level}(c, k) \wedge \mathsf{localTo}(c', o') \wedge \mathsf{level}(c', k) \Rightarrow \mathsf{subclassOf}(c, c') \quad \text{(A12)}$$

According to (A13) when c is a subclass of c' and c has a local metaclass m representing level k and c' has a local metaclass m' representing level k, then m is a subclass of m'. This applies also to classes of higher orders. For example, `CarModel` is a subclass of `ProductModel`, its local metaclasses for level Individual, namely `CarIndividualClassByModel` and `ProductIndividualClassByModel` are, hence, in a subclassOf relationship.

$$\forall c, c', m, m', k : \mathsf{subclassOf}(c, c') \wedge \mathsf{localTo}(m, c) \wedge \mathsf{level}(m, k) \wedge \mathsf{localTo}(m', c') \wedge \mathsf{level}(m', k) \Rightarrow \mathsf{subclassOf}(m, m') \quad \text{(A13)}$$

Finally, the *members* of a class c (of any order) are defined as the instances of that class together with instances of direct or indirect subclasses of c.

5.5 Aligning DDO with MLT

To illustrate the semantics of multi-level models induced by domain object hierarchies in terms of the multi-level theory MLT [8] we now define the derivation of *partition* and *subordinate to* relationships. This is for informative purposes and does not add to the semantics of the approach. The `partitions` and `subordinateTo` relations for the running example are shown in Fig. 10.

An induced fact $\mathsf{partitions}(m, c)$ says that metaclass m partitions class c, i.e., every instance of m is a subclass of c, and, further, every member of c is member of exactly one instance of m. An induced fact $\mathsf{subordinateTo}(m, m')$ says that metaclass m is subordinate to m' which means that every instance of m is subclass of some instance of m'.

According to (A14), when a class c representing level k is local to a domain object o and another class c', also local to o, has a local metaclass m also representing level k, then m partitions c. In this case, c is also referred to as the base class of m.

$$\forall c, k, o, c', m : \mathsf{level}(c, k) \wedge \mathsf{localTo}(c, o) \wedge \mathsf{localTo}(c', o) \wedge \mathsf{localTo}(m, c') \wedge \mathsf{level}(m, k) \Rightarrow \mathsf{partitions}(m, c) \quad \text{(A14)}$$

(A14) only derives metaclasses that partition first-order classes. Higher-order partitioning metaclasses are derived according to (A15) as follows. When a metaclass m partitions a class c and both have local metaclasses m' and m'', respectively, which both represent level k, then m' partitions m''.

$$\forall m, c, m', m'', k : \mathsf{partitions}(m, c) \wedge \mathsf{localTo}(m', m) \wedge \mathsf{level}(m', k) \wedge \mathsf{localTo}(m'', c) \wedge \mathsf{level}(m'', k) \Rightarrow \mathsf{partitions}(m', m'') \quad \text{(A15)}$$

The subordinateTo relationship, according to MLT [8], is a relationship between partitioning metaclasses with the same base class. When a partitioning metaclass is subordinate to another partitioning metaclass then every instance of the former is a subclass of some instance of the latter. Now let us define how subordinateTo relationships are induced from domain object hierarchies. According to (A16), when a domain object o has two local classes c and c' with c being directly under c' in the part-whole hierarchy schema, and c and c' have both a local metaclass, m and m', respectively, which both represent level k, then m is subordinate to m'.

$$\begin{aligned}&\forall o, c, c', m, m, k : \mathsf{localTo}(c, o) \wedge \mathsf{localTo}(c', o)\\&\quad\wedge \mathsf{localTo}(m, c) \wedge \mathsf{level}(m, k)\\&\quad\wedge \mathsf{localTo}(m', c') \wedge \mathsf{level}(m', k)\\&\quad\wedge \mathsf{partOfSchema}(c, c') \Rightarrow \mathsf{subordinateTo}(m, m')\end{aligned} \tag{A16}$$

Similar to (A14) above, (A16) only applies to metaclasses partitioning first-order classes. Subordination among higher-order classes is derived according to (A17) as follows. When a metaclass m is subordinate to metaclass m' and both have local metaclasses m'' and m''', respectively, which both represent level k, then m'' partitions m'''.

$$\begin{aligned}&\forall m, m', m'', m''', k : \mathsf{subordinateTo}(m, m')\\&\quad\wedge \mathsf{localTo}(m'', m) \wedge \mathsf{level}(m'', k)\\&\quad\wedge \mathsf{localTo}(m''', m') \wedge \mathsf{level}(m''', k)\\&\quad\Rightarrow \mathsf{subordinateTo}(m'', m''')\end{aligned} \tag{A17}$$

5.6 Discussion

Let us briefly assess the induced multi-level model with regard to the desiderata of Sect. 3.1. The approach is *non-disruptive* (+) as the part-whole hierarchy remains intact and the multi-level structures act as add-on. *Conceptual clarity* (+) is facilitated by decomposing domain objects into their different facets. *Compactness* (+) is reached through the nesting of induced classes and metaclasses with domain objects acting as roots of localTo-hierarchies. The central part-whole hierarchy *factors out commonalities* (+) among the multiple generalization and multi-level instantiation hierarchies. Avoiding custom constraints (+) is facilitated by having fine-grained classes and metaclasses readily available which spares the modeler from writing derivation rules by hand.

6 Modeling with deep domain objects

In this section we introduce and exemplify modeling with deep domain objects along the modeling challenges set out in Sect. 3.

A deep domain object is a domain object together with an aggregated and schematic description of its descendant domain objects together with regulations that govern how the schema is to be specialized along the hierarchy. Every domain object (apart from objects at the bottom level) has an induced multi-level schema characterizing the sub-hierarchy rooted in that domain object.

In the simple case of a domain object hierarchy with a totally ordered set of levels, a deep domain object is similar to a DDM clabject but with the various class/superclass/metaclass facets of the DDM clabject now represented explicitly by classes and metaclasses 'nested' inside the domain object and with support for providing aggregated information with these local meta* classes. These local classes and metaclasses are induced automatically, together with the multi-level dependencies connecting classes which are local to the same object and the dependencies connecting classes local to objects that are in a direct hierarchical relationship in the domain object hierarchy. Furthermore, by encapsulating local classes and metaclasses in deep domain objects, we retain one of the promises of dual deep modeling, namely to avoid the cluttering of models with a myriad of classes and metaclasses.

6.1 Solving challenge 1: specialization along the hierarchy

In this subsection we explain how the modeling tasks presented in Sect. 3.2 are solved with deep domain objects (see Fig. 12).

C1.1 Range refinements. Induced local classes facilitate the refinement of the range of a property p for the descendants of a domain object o at level k to descendants of a domain object o' at level k'.

For example, with local class `CarModel` the range of property `productMgr` is refined to local class `Vehicles Employee`, that is, product models of product category `Car` may have as product manager only employees of sales division `Vehicles`.

With local class `CarIndividual` the range of property `soldAt` is refined to local class `VehiclesOutlet`, that is, individual cars may only be sold at sales outlets of the `Vehicles` sales division.

C1.2 Extending the schema. With local classes the modeler is able to extend the schema of descendants of a domain object at a particular level by introducing additional properties.

For example, with local class `CarModel` the modeler extends the schema with an additional property `engineType` with range `EngineType` (from the Engine hierarchy). With local class `CarIndividual` an additional property `engine` with range `EngineIndividual` is introduced.

C1.3 Shared values. With local classes the modeler may also specify a shared value for a property of descendant objects at a certain level.

For example, with local class `CarModel` the modeler specifies a shared value of 20% for property `vatRate`—specifying that every car model has exactly that value for property `vatRate`. With local class `CarIndividual` the `vatRate` value of 20% is fixed for all individual cars.

Assessment of DDO solution to Challenge 1 Let us discuss the quality of the DDO solution with regard to the desiderata (see Sect. 3.1). The solution is *non-disruptive* (+) since the original conceptual model remains intact. *Conceptual clarity* (+) is reached by modeling properties that charac-

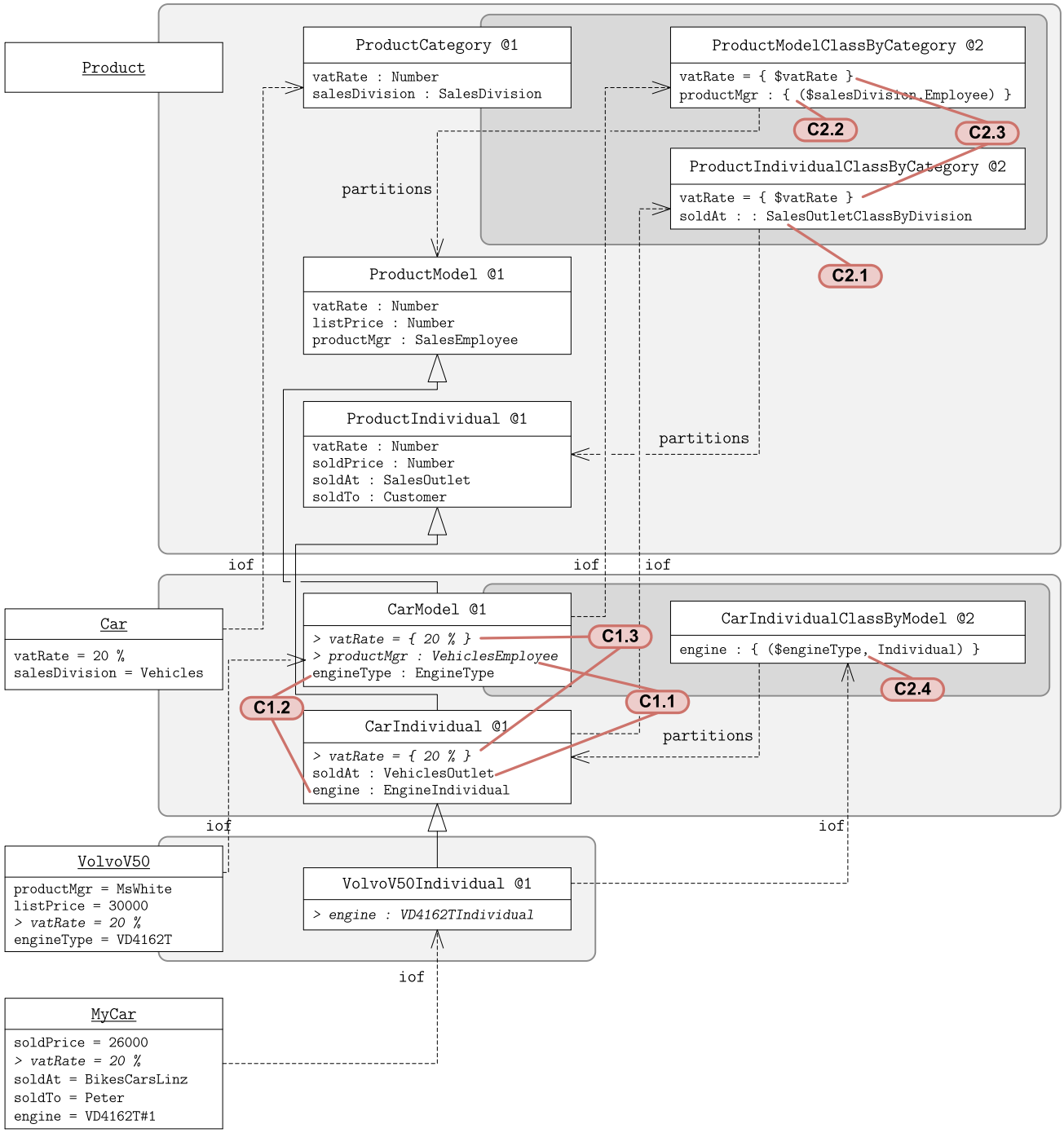


Fig. 12 Solving Challenges 1 and 2 using the DDO approach. Property statements inferred by metaschema statements are shown with '>' and in italics. A property p's metarange m is denoted as p : : m. A propagate-as-shared-value assertion, denoted as $p = \{\$p'\}$, specifies the propagation of the p' value of the domain object of the member class as shared value of the member class' property p. A propagate-as-range assertion, denoted as p : $\{(\$p',k)\}$, specifies the propagation of the p' value of the domain object of the member class as range (together with level k) of the member class' property p. Deep domain objects only shown with the induced classes and metaclasses relevant for that challenge

terize descendants at a particular levels with local classes. *Compactness* (+) is reached since the specializations regarding the sub-hierarchy rooted in domain object `Car` are all modeled with model elements local to `Car`. With regard to *factoring out commonalities* (+), the modeler is not forced to represent the same modeling information twice. The solution does not come with *custom constraints* (+).

6.2 Solving challenge 2: regulating specialization with metaclasses

Domain object hierarchies typically have a rather homogeneous hierarchical structure across their sub-hierarchies. This degree of homogeneity will typically also be reflected in the schema refinements. Metaschema assertions can enforce a certain degree of homogeneity and can also automate schema refinements.

Before solving Challenge 2, let us have a closer look at the meaning of local metaclasses from the perspective of domain object hierarchies.

A metaclass, e.g., `ProductIndividualClassByCategory` represents a level, e.g., Individual, which we refer to as its *member-member level*. The metaclass is local to a class, e.g., `ProductCategory`, which, in turn, represents another level, e.g., Category, which we refer to as the metaclass' *member level*, and is local to a domain object, e.g., `Product`. For a particular state of the database, a metaclass represents a metaset, namely the set of descendants of the domain object, e.g., `Product`, at the metaclass' member-member level, e.g., Individual, grouped by the metaclass' member-level, e.g., Category. Its member classes are induced subclasses local to domain objects at the metaclass' member level.

The modeling tasks concerning regulating specialization are solved with deep domain objects (see Fig. 12) as follows.
C2.1 Metarange. A metarange assertion specifies an obligation for members (which are classes) of the metaclass to refine the schema of a property to a certain hierarchy level. The modeler is able to specify with a (local metaclass of a) higher-level object a constraint that a certain refinement should be done by all descendant domain objects at a certain level (namely by local classes of descendant domain objects at that level).

For example, the metaclass `ProductIndividualClassByCategory` further defines `SalesOutletClassByDivision` as metarange of its member-member property `soldAt`, denoted as `soldAt : : SalesOutletClassByDivision`, meaning that its member classes at the `Category` level, e.g., `CarIndividual` must refine the range of `soldAt` to outlets that belong to a particular sales division.
C2.2 Propagate property value as range. A *propagate-as-range* assertion tells members of the metaclass to use a property value of the domain object they are local to, to refine the range of another property.

Using propagate-as-range assertions the modeler can generically specify the refinement of property ranges for domain objects at lower levels (i.e., the metaclass' member-member level) by property values of domain objects at intermediate levels (i.e., the metaclass' member level).

For example, the metaclass `ProductModelClassByCategory` has a propagate-as-range assertion for its member-member property `productMgr`, denoted as `productMgr : {($salesDivision,Employee)}`, defining that its member classes at the category level, such as `CarModel`, take the value, e.g., `Vehicles`, of property `salesDivision` of the product category, e.g., `Car`, to refine the range of the property to employees of that sales division, e.g. by local class `VehiclesEmployee`.
C2.3 Propagate property value as shared value. A *propagate-as-shared-value* assertion tells members of the metaclass to use a property value of the domain object they are local to, to define a shared value for the metaclass' member-member property.

Using propagate-as-shared-value assertion the modeler can generically define the propagation of property values from domain objects at an intermediate level (i.e., the metaclass' member level) to domain objects at lower levels (i.e., the metaclass' member-member level) as shared values.

For example, the metaclass `ProductModelClassByCategory` has a propagate-as-shared-value assertion for its member-member property `vatRate`, denoted as `vatRate = { $vatRate }`, defining that its member classes, such as `CarModel`, use the `vatRate` given with the product category they are local to, e.g., `Car`, as shared value for their member property `vatRate`. Metaclass `ProductIndividualClassByCategory` has a similar assertion, specifying the propagation of the `vatRate` to the individual level.
C2.4 Regulating specialization in sub-hierarchies. With local metaclasses the modeler is able to specify such regulations also only for sub-hierarchies.

For example, a propagate-as-range assertion given with metaclass `CarIndividualClassByModel` specifies that the engine type, e.g., `VD4162T`, given with a particular car model, such as `VolvoV50`, is used to refine the range for the individual class of that car model, e.g., in class `VolvoV50Individual` the range for property `engine` gets automatically refined to class `VD4162TIndividual` (the class of individual engines of engine type `VD4162T`).
Assessment No customs constraints (+) are necessary and the solution is non-disruptive (+). Conceptual clarity (+) is high because each metaschema assertion is made with a model element that specifically represents the two affected levels in a domain object hierarchy or sub-hierarchy. Compactness (+) is reached since the metaschema assertions are made with

metaclasses which are tied together with a domain object. The solution avoids custom constraints (+). Regarding *factoring out commonalities* (+), there is no need to model some information twice.

6.3 Solving challenge 3: aggregated information

In the DDO approach, every class has associated its power type. A class' power type is used to define properties that are to be instantiated by the class and also by the class' direct and indirect subclasses, which are all, by definition, members of the power type. Note, in the DDO approach we currently only have use for one layer of power types, that is why there are no power types of power types.

To solve modeling challenge 3 the modeler defines (see Fig. 13) aggregated properties `avgListPrice`, `avgSoldPrice`, and `minAvgSoldPricePerModel`, with the power types of `ProductModel`, `ProductIndividual`, and `ProductIndividualClassByModel`, respectively.

By using the power type of a class representing a certain level (e.g., Model) for the definition of aggregated properties, the modeler makes sure that the aggregated information is available with domain objects above that level without the need to define the aggregated property multiple times.

Aggregated information is also available for the domain object hierarchy as a whole.

The class with which the aggregated information is provided represents the set of objects from which the aggregated information is calculated. The metaclass with which a two-step aggregated information is provided represents the set of objects and its partitioning into a set of set of objects from which the aggregated information is calculated.

C3.1 One-step aggregation. Simple aggregated information, i.e., aggregated properties like `avgSoldPrice` and `avgListPrice` are defined with a powertype of a first-order local class and the aggregated vlaues are provided with first-order local classes.

C3.2 Multi-step aggregation. Two-step aggregated information (e.g., `minAvgSoldPricePerModel`) is provided with second-order classes, and so forth (no example given for three step aggregation).

C3.3 Aggregated information for the whole hierarchy. By defining aggregated properties with a powertype of a class that is local to a root object, the aggregated information is calculated also for the hierarchy as a whole.

Assessment. No custom constraints (+) or complex queries are necessary. Compactness (+) and conceptual clarity (+) is reached since the aggregated information is provided with the class or metaclass it describes which is also tied together with the domain object. Concerning factoring out commonalities (+), there is no need to define an aggregated property multiple times.

7 Proof-of-concept prototype

The deductive rules introduced in Sect. 5 are realized in F-Logic (see Appendix A) together with the formalization/implementation of the complete DDO approach. The prototype can be used to query induced multi-level models and property values, including aggregated information and shared values, and property ranges, including those propagated from higher-level objects. This is demonstrated by the query results shown in Figs. 14, 15, 16, and 17. One further query result is provided in the appendix. Formating query results in tabular form is implemented in a small Java program which interacts with Flora-2 via command line.

In addition to the derivation of local classes and metaclasses and their multi-level dependencies, the prototype covers:

- Specialization along the hierarchy with range refinement, schema extension and shared values.
- Regulating specialization with metaclasses by metarange assertions, propagate-as-range assertions, and propagate-as-shared-value assertions for domain object hierarchies as a whole and also for sub-hierarchies.
- Modeling aggregated properties with powertypes and providing respective aggregated information with local classes and metaclasses.
- Structural conformance checks.

Conformance checks are also realized as deductive rules which fill predicate `error/2`, which can be queried for violations of the structural conformance rules.

The functionality of the prototype is demonstrated with the running example started in Sect. 2 and further developed in Sect. 6. The fully worked example is given in Appendix B.

Note, in the F-Logic implementation we use custom properties `subclassOf` and `iof` instead of the built-in notation used by F-Logic ('`::`' and '`:`'). This is to avoid an unintended strong integration into F-Logic, since we do not conceive the implementation as an add-on to F-Logic but rather only as a validation of the approach. Nevertheless, to facilitate preliminary integration with F-Logic's built-in instantiation and subclassing we have to consider them in the realization of property range constraints. We further consider in the F-Logic implementation that a class c' in the formalization may be a data type or a 'normal' class from the global schema or a local class from some domain object hierarchy.

The proof-of-concept prototype comes with four aggregation operators, namely SUM, MIN, MAX, and AVG (see Appendix A). In the definition of the aggregated property in the metaclass (typically a powertype), these operators take as parameter a member property of their base class. The aggregation operation is then evaluated with regard to an instance

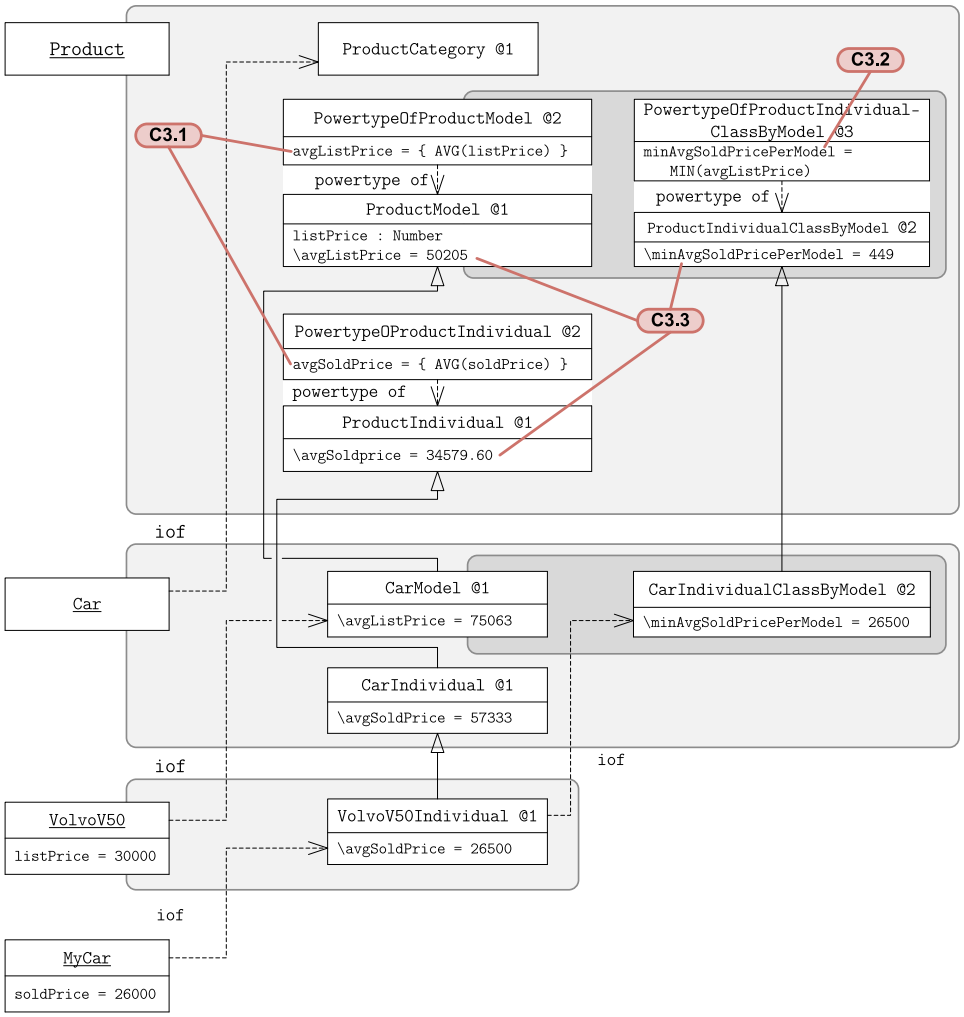


Fig. 13 Solving Challenge 3 (aggregated information) using power types of classes and of metaclasses to define aggregated properties

of the powertype, that is, with regard to a subclass of the base class, and aggregates all members of that subclass.

Concerning performance we have conducted very preliminary experiments on a HP EliteBook 850 G2 with Intel(R) Core(TM) i7-5600U CPU with 2.60GH with two kernels with 16 GB working memory and running Windows 10 Pro. The time measurements are based on 10 consecutive runs of a custom-made Java program that starts the Flora engine, interacts with the Flora engine via command line, and which transforms the query results to tabular format.

We have experimented with two variants. The first variant (see Appendix A.2) uses structured class IDs for local classes and powertypes, e.g., `k(k(Product,Model),Individual)`. The second variant (see Appendix A.3) generates class names as used throughout the paper, e.g., `ProductIndividualClassByModel`. With the first variant, starting the Flora2 engine and loading the code takes on average 0.66 sec, the evaluation of the five queries (as shown in Figures 14–18) takes together on average 1.14 sec. With the second variant, startup took on average 0.81 sec and query evaluation on average 3.52 s.

8 Related work

In multi-level modeling, the *clabject* [1] allows meta-modeling with arbitrarily many levels of instantiation. *Deep characterization* allows a clabject to specify the schema of members at arbitrarily deeper levels of instantiation, it can specify attributes to be instantiated by its members or by the members of its members, and so forth.

Telos [27] was one of the first approaches that allowed meta-modeling with an unbounded number of instantiation levels. In Telos, classes and properties are also objects, and objects are instances of other objects. ConceptBase [17] is an implementation of Telos based on the O-Telos [18] dialect.

VODAK [21] was one of the first systems to support deep characterization, although limited to three levels of instantiation. VODAK distinguishes three kinds of objects,

```
?- ?[localToDO->?DOMAINOBJECT, order->?i], ?i>0, ?CLASS [ order->?i, localToDO->?DOMAINOBJECT ], (?CLASS[ subclassOf -> ?subclassOf ];
(\naf \exists(?_x)^(?CLASS[ subclassOf -> ?_x]), ?subclassOf = nil)),(?CLASS[ partOfSchema -> ?partOfSchema ]; (\naf
\exists(?_x)^(?CLASS[ partOfSchema -> ?_x]), ?partOfSchema = nil)).
30 solution(s) in 0.140 seconds; elapsed time = 0.155
Yes
```

no	?DOMAINOBJECT	?i	?CLASS	?subclassOf	?partOfSchema
1	Car	1	CarIndividual	ProductIndividual	CarModel
2	Car	1	CarModel	ProductModel	nil
3	Car	2	CarIndividualClassByModel	ProductIndividualClassByModel	nil
4	Electronics	1	ElectronicsEmployee	SalesEmployee	nil
5	Electronics	1	ElectronicsOutlet	SalesOutlet	nil
6	Engine	1	EngineIndividual	nil	EngineType
7	Engine	1	EngineType	nil	nil
8	Engine	2	EngineIndividualClassByType	nil	nil
9	IPhone6	1	IPhone6Individual	PhoneIndividual	nil
10	P3LTwinTurbo	1	P3LTwinTurboIndividual	EngineIndividual	nil
11	P911	1	P911Individual	CarIndividual	nil
12	Phone	1	PhoneIndividual	ProductIndividual	PhoneModel
13	Phone	1	PhoneModel	ProductModel	nil
14	Phone	2	PhoneIndividualClassByModel	ProductIndividualClassByModel	nil
15	Product	1	ProductCategory	nil	nil
16	Product	1	ProductIndividual	nil	ProductModel
17	Product	1	ProductModel	nil	ProductCategory
18	Product	2	ProductIndividualClassByCategory	nil	ProductModelClassByCategory
19	Product	2	ProductIndividualClassByModel	nil	nil
20	Product	2	ProductModelClassByCategory	nil	nil
21	Product	3	ProductIndividualMetaclassByModelByCategory	nil	nil
22	Sales	1	SalesDivision	nil	nil
23	Sales	1	SalesEmployee	nil	SalesDivision
24	Sales	1	SalesOutlet	nil	SalesDivision
25	Sales	2	SalesEmployeeClassByDivision	nil	nil
26	Sales	2	SalesOutletClassByDivision	nil	nil
27	V50	1	V50Individual	CarIndividual	nil
28	Vd4162T	1	Vd4162TIndividual	EngineIndividual	nil
29	Vehicles	1	VehiclesEmployee	SalesEmployee	nil
30	Vehicles	1	VehiclesOutlet	SalesOutlet	nil

Fig. 14 Result of querying all classes in the multilevel model, together with, if existing, their direct superclass, and, if existing, their parent in the part-whole hierarchy

```
?- ?[localToDO->?DOMAINOBJECT,order->?order ], ?METACLASS [localToDO->?DOMAINOBJECT, order->?order, partitions->?partitions ], (?METACLASS[
subordinateTo -> ?subordinateTo ]; (\naf \exists(?_x)^(?METACLASS[ subordinateTo -> ?_x]), ?subordinateTo = nil)).
9 solution(s) in 0.047 seconds; elapsed time = 0.058
Yes
```

no	?DOMAINOBJECT	?order	?METACLASS	?partitions	?subordinateTo
1	Car	2	CarIndividualClassByModel	CarIndividual	nil
2	Engine	2	EngineIndividualClassByType	EngineIndividual	nil
3	Phone	2	PhoneIndividualClassByModel	PhoneIndividual	nil
4	Product	2	ProductIndividualClassByCategory	ProductIndividual	nil
5	Product	2	ProductIndividualClassByModel	ProductIndividual	ProductIndividualClassByCategory
6	Product	2	ProductModelClassByCategory	ProductModel	nil
7	Product	3	ProductIndividualMetaclassByModelByCategory	ProductIndividualClassByModel	nil
8	Sales	2	SalesEmployeeClassByDivision	SalesEmployee	nil
9	Sales	2	SalesOutletClassByDivision	SalesOutlet	nil

Fig. 15 Result of querying all partitioning metaclasses in the multilevel model, together with the base class they are partitioning and, if existing, the metaclass they are subordinate to

namely individual objects, classes, and metaclasses, as well as types, which describe the schema of objects in terms of structure and behaviour. An object has an own-type which describes the object's own schema. A class or metaclass further has a member-type which describes the schema of its members. A metaclass further has a member-member-type which describes the schema of the members of its members, realizing a limited form of deep characterization. An object instantiates its own-type, which in turn is a specialization of the member-type of the object's class. The member-type of a class is a specialization of the member-member-type of the metaclass of the class. In this way, the relationship between an object and its class encompasses instantiation and specialization.

Potency-based deep instantiation [2] goes beyond VODAK by allowing to specify the schema of member objects at arbitrarily deeper levels of instantiation by assigning a potency to properties.

```
?- ?DOMAINOBJECT : DomainObject, ?object[value(?PROPERTY)->?VALUE, localToDO->?DOMAINOBJECT].
55 solution(s) in 0.047 seconds; elapsed time = 0.042
Yes
```

no	?DOMAINOBJECT	?object	?PROPERTY	?VALUE
1	AdasCar	AdasCar	engine	Vd4162T_2
2	AdasCar	AdasCar	maxSpeed	190
3	AdasCar	AdasCar	soldAt	CarStoreVienna
4	AdasCar	AdasCar	soldPrice	27000
5	AdasCar	AdasCar	soldTo	Ada
6	AdasCar	AdasCar	vatRate	0.2
7	Car	Car	distributedBy	Vehicles
8	Car	Car	vatRate	0.2
9	Car	CarIndividual	avgSoldPrice	57333.3333333333
10	Car	CarIndividualClassByModel	maxAvgSoldPricePerModel	119000.0
11	Car	CarModel	avgListPrice	75062.5
12	IPhone6	IPhone6	listPrice	490
13	IPhone6	IPhone6	productMgr	MsMuller
14	IPhone6	IPhone6	vatRate	0.16
15	IPhone6	IPhone6Individual	avgSoldPrice	449.0
16	JoesPhone	JoesPhone	soldAt	EStoreVienna
17	JoesPhone	JoesPhone	soldPrice	459
18	JoesPhone	JoesPhone	soldTo	Joe
19	JoesPhone	JoesPhone	vatRate	0.16
20	MiasCar	MiasCar	engine	P3TT_1
21	MiasCar	MiasCar	maxSpeed	250
22	MiasCar	MiasCar	soldAt	BikesCarsLinz
23	MiasCar	MiasCar	soldPrice	119000
24	MiasCar	MiasCar	soldTo	Mia
25	MiasCar	MiasCar	vatRate	0.2
26	MiasPhone	MiasPhone	soldAt	EStoreVienna
27	MiasPhone	MiasPhone	soldPrice	439
28	MiasPhone	MiasPhone	soldTo	Mia
29	MiasPhone	MiasPhone	vatRate	0.16
30	MyCar	MyCar	engine	Vd4162T_1
31	MyCar	MyCar	maxSpeed	190
32	MyCar	MyCar	soldAt	BikesCarsLinz
33	MyCar	MyCar	soldPrice	26000
34	MyCar	MyCar	soldTo	Me
35	MyCar	MyCar	vatRate	0.2
36	P911	P911	engineType	P3LTwinTurbo
37	P911	P911	listPrice	120125
38	P911	P911	productMgr	MsJones
39	P911	P911	topSpeed	250
40	P911	P911	vatRate	0.2
41	P911	P911Individual	avgSoldPrice	119000.0
42	Phone	Phone	distributedBy	Electronics
43	Phone	Phone	vatRate	0.16
44	Phone	PhoneIndividual	avgSoldPrice	449.0
45	Phone	PhoneIndividualClassByModel	maxAvgSoldPricePerModel	449.0
46	Phone	PhoneModel	avgListPrice	490.0
47	Product	ProductIndividual	avgSoldPrice	34579.6
48	Product	ProductIndividualClassByModel	maxAvgSoldPricePerModel	119000.0
49	Product	ProductModel	avgListPrice	50205.0
50	V50	V50	engineType	Vd4162T
51	V50	V50	listPrice	30000
52	V50	V50	productMgr	MrWhite
53	V50	V50	topSpeed	190
54	V50	V50	vatRate	0.2
55	V50	V50Individual	avgSoldPrice	26500.0

Fig. 16 Result of querying property values (including aggregated property values, aggregated from descendants at lower levels, and values that were propagated as shared values from higher-level objects). By having both the object to which the property value belongs directly as well as the domain object to which that object belongs, the DDO approach combines compactness and conceptual clarity. Note: different background colors were added manually

```
?- ?DOMAINOBJECT : DomainObject, ?CLASS[range(?PROPERTY)->?RANGE, localToDO-
>?DOMAINOBJECT].
19 solution(s) in 0.000 seconds; elapsed time = 0.003
Yes
```

no	?DOMAINOBJECT	?CLASS	?PROPERTY	?RANGE
1	Car	CarIndividual	engine	EngineIndividual
2	Car	CarIndividual	maxSpeed	\number
3	Car	CarIndividual	soldAt	VehiclesOutlet
4	Car	CarModel	engineType	EngineType
5	Car	CarModel	productMgr	VehiclesEmployee
6	Car	CarModel	topSpeed	\number
7	P911	P911Individual	engine	P3LTwinTurboIndividual
8	Phone	PhoneIndividual	soldAt	ElectronicsOutlet
9	Phone	PhoneModel	productMgr	ElectronicsEmployee
10	Product	ProductCategory	distributedBy	SalesDivision
11	Product	ProductCategory	vatRate	\number
12	Product	ProductIndividual	soldAt	SalesOutlet
13	Product	ProductIndividual	soldPrice	\number
14	Product	ProductIndividual	soldTo	Customer
15	Product	ProductIndividual	vatRate	\number
16	Product	ProductModel	listPrice	\number
17	Product	ProductModel	productMgr	SalesEmployee
18	Product	ProductModel	vatRate	\number
19	V50	V50Individual	engine	Vd4162TIndividual

Fig. 17 Result of querying property ranges (including property ranges propagated according to range-propagation assertions). By having both the class to which the property range belongs directly as well as the domain object to which that class is local to, the DDO approach combines compactness and conceptual clarity. Note: different background colors and horizontal lines were added manually

The orthogonal classication architecture distinguishes linguistic and ontological metamodeling [2,3]. Linguistic metamodeling is about the modeling of elements of the modeling language, e.g., `Domain Class` and `Domain Object`, and their instantiation by model elements, e.g, `ProductModel` and `VolvoV50`, respectively. Ontological metamodeling is about modeling of individuals, classes and metaclasses from the domain of interest and their ontological instance-of relationships, e.g., `VolvoV50` is ontological instance of `CarModel`. In our approach, ontological instantiation is divided into two parts, domain object `VolvoV50` is, rst, an instance of domain class `ProductModel` representing level Model, and, second, under domain object `Car`, so that it is nally an ontological instance of `CarModel`.

With regard to Kühne's level cohesion and level segregation principles [22], classification levels in DDO hierarchies are *order-synchronized*, yet the organization in classification levels is second to the organization along domain object hierarchies.

The DDO pattern is related to *materialization* [33] which also combines instantiation and specialization. With materialization, a class of more concrete objects, e.g., ProductModel, may be related to a class of more abstract objects, e.g., ProductCategory. Each member, e.g., VolvoV50, of the former materializes a member, e.g, Car, of the latter. Materialization supports level-crossing property propagation.

The power type was introduced to conceptual modeling by Odell [31] as a class whose members are subclasses of another class.Power types were applied and evolved by Gonzalez-Perez and Henderson-Sellers [15] and analyzed in full detail by Carvalho et al. [8,10]. The *partitions* relationship which underlies our approach is a specific form of a power type relationship in the sense of Odell. Our notion of power types was introduced by Cardelli [7] and revisited by Carvalho et al. [8].

Some of the ideas underlying the current approach were already implicitly realized in the M-Object approach [28] but lacking the foundation in traditional object-oriented modeling and lacking the explicit induction of meta*-classes. An m-object, e.g., `Car`, has a set of totallyordered named levels, e.g., `Category`, `Model`, and `Individual`. Each level of an m-object implicitly defines a class with the top-level of an m-object implicitly defining a singleton class with the mob-

ject itself as member. When a more concrete mobject, e.g., `VolvoV50`, concretizes a more abstract m-object, e.g., `Car`, then the classes implicitly defined by the levels of the more concrete m-object are specializations of the classes implicitly defined by the levels of the more abstract m-object, and the more concrete m-object is a member of the class implicitly defined by the second-top level of the more abstract m-object. Other than in the DDO approach, m-objects are not related by properties but by multi-level relationships instantiates its top-level class, i.e., its own-type. An m-object, e.g., `Car`, may concretize another m-object, e.g., `Product`. The concretization relationship between m-objects has an instantiation and specialization facet. The concretizing m-object instantiates the concretized m-object's second-level class and specializes the concretized m-object's other classes. M-relationships relate m-objects at various levels.

The practical relevance of multi-level modeling was demonstrated by examining the application of multilevel modeling patterns in existing metamodels from different domains [23]. We believe that our approach fundamentally extends the practical relevance of multi-level modeling to modeling scenarios where domain objects are organized in part-of hierarchies (or similar) and where the goal is to refine and extend the schema along that hierarchy.

There is an ongoing discussion about the advantages and drawbacks of multi-level modeling with clabjects [5,11]. Various authors [9,34] argue that there is trade-off between compactness (reduction of model size) and understandability (semantic clarity). We claim that with our current approach both goals, namely semantic clarity and reduction of model size (of the model presented to the modeler), are met, first, by the approach's foundation in MLT and, second, by inducing multi-level structures and showing them only on demand.

Multilevel Coupled Model Transformations (MCMTs) [25] are an expressive approach for specifying level-crossing constraints. It remains to be analyzed how MCMTs can be employed on top of our approach or as an extension to our simple expression mechanism. Balaban et al. [6] introduce an executable multi-level modeling approach facilitating inter-level constraints, rules, and queries.

The multi-level theory (MLT) [8–10] improves semantic clarity through ontologically well-founded multi-level modeling constructs aligned with Guizzardi's Unified Foundational Ontology UFO [16]. The DDO approach is aligned in parts with MLT but does not support all cross-level structural relations identified in [8], in addition to *power type of* and *partioning*, they also propose a more relaxed relation called *categorization* which may be *disjoint* and/or *complete*. The partitions relation (which is fundamental to our approach) is a disjoint and complete categorization relation. It remains open to further analysis how the classes in the global model (which are classes without super-classes) relate to the notion of substance sortals as discussed by Guizzardi [16]).

Local object classes were proposed by Kappel and Schrefl [20] as a means to represent local referential integrity. There, a local object class 'is a set of objects belonging exclusively to some composite object", and the "database is considered an object itself" and "every object class is a local object class".

9 Conclusion

In this paper we have introduced the DDO approach, which may serve as the nucleus of multi-level modeling approaches that build on traditionally modeled domain object hierarchies. We have defined and implemented the approach in F-Logic/Flora-2. We have exemplified all constructs by a fully worked example. We have assessed the approach with regard to a set of desiderata together with a set of to-be supported modeling tasks. The proof-of-concept implementation of the approach and the fully worked example in F-Logic are provided in the Appendix and are made available as free software.

With the prototype, we have shown the feasibility of the approach and demonstrated its use for querying induced multi-level models. Qualitative evaluation of the approach in terms of usability and comprehensibility remains for future work, as well as systematic performance studies. In future work we will also explore the remaining design space for a full-fledged DDO-based multi-level modeling approach, especially regarding the multi-level modeling of properties and associations.

Funding Open access funding provided by Johannes Kepler University Linz.

A Implementation of DDO in F-logic

The implementation of the DDO approach in F-Logic was developed and tested with Flora-2/ErgoLite Reasoner 2.0 (Pyrus nivalis) of 2018-10-14. The prototype implementation is available as free software under https://github.

com/bneumayr/DDO. The Flora-2/ErgoLite system is necessary to run the prototype and is available at http://flora.sourceforge.net/. The following code listing contains the complete implementation of the formalization of DDO.

A.1 Rules for preparing the hierarchy `prepare.flr`

```
// A1: The hierarchy is treated as domain object at the
     generic Root level
?o : DomainObject :-
  ?o : Hierarchy .

?o [ level -> Root ] :-
  ?o : Hierarchy.

// A2: Domain objects at a top level of the part-whole
     hierarchy are direct parts of root object
?o [ partOf -> ?h ] :-
  ?o [ instanceOf -> ?c ],
  ?c [ localTo -> ?h ],
  \naf exists(?c2)^(?c [partOfSchema->?c2] ).

// A3: Domain objects are annotated with their level
?o [ level -> ?k ] :-
  ?o [ instanceOf -> ?c ],
  ?c [ level -> ?k ].

// A4: Domain objects have order 0
?o [ order -> 0 ] :-
  ?o : DomainObject.
```

A.2 Rules for creating classes with structured IDs `class_creation_structures.flr`

With regard to the naming of derived local classes and powertypes there are two variants, one working with Prolog structures as class IDs (e.g., `k(Car,Model)`, `k(k(Car,Model),` and `pow(k(k(Car,Model),Individual))`), and one creating class names (e.g., `CarModel`, `CarIndividualClas` and `PowertypeOfCarIndividualClassByModel`) (see A.3). In F-Logic, working with structured IDs is simpler and faster in comparison to generating class names by string concatenation.

```
// A6 Create first-order local classes
k(?o,?k) [ localTo -> ?o, level -> ?k ] :-
  ?o [ iof -> ?c ],
  ?[ partOfSchemaT -> ?c, level -> ?k ].

// A7 Create higher-order classes up to maxOrder
k(?c,?k) [ localTo -> ?c, level -> ?k ] :-
  ?_c [ partOfSchemaT -> ?c, level -> ?k ] ,
  ?c.order < config.maxOrder.

// Every local class (of any order) has a powertype
?c [ powertype -> pow(?c) ] :-
  ?c [ localTo -> ?].
```

A.3 Alternative: rules for creating named classes `class_creation_names.flr`

The following rules create local classes and powertypes with names, e.g., `CarModel` and `CarIndividual`, as used throughout the paper. These names are generated by string concatenation which comes with performance overhead.

```
// Create first order local classes
?new [ localTo -> ?o, level -> ?k] :-
  ?o [ iof -> ?c ],
  ?[ partOfSchemaT -> ?c, level -> ?k ],
  name ( ?o.localToDO, ?o.orderName, ?k, ?o.by,
       ?new ).

// Create higher-order local classes up to maxOrder
?new [ localTo -> ?c, level -> ?k ] :-
  ?_c [ partOfSchemaT -> ?c, level -> ?k ],
  ?c.order < config.maxOrder ,
  name ( ?c.localToDO, ?c.orderName, ?k, ?c.by,
       ?new ) .

// Every local class (of any order) has a powertype
?c [ powertype -> ?pow ] :-
  ?c [ localTo -> ?],
  \symbol[ concat([PowertypeOf,?c]) -> ?pow
       ]@\basetype.

 Create class name by string concatenation
name ( ?do, ?orderName, ?level, ?by, ?name) :-
  \symbol[ concat([?do,?level,?orderName,?by])
       -> ?name ]@\basetype.

 Associate order numbers with Names
orderToName(0,'').
orderToName(1,'Class').
orderToName(2,'Metaclass').
orderToName(3,'Metametaclass').
orderToName(4,'Metametametaclass').

 Derive the order name for every object
?o [ orderName -> ?oName ] :-
  orderToName(?o.order,?oName) .

 Derive the partioned-by-Part of metaclass names
?o [ by -> '' ] :-
  ?o : DomainObject .
?c [ by -> ?by ] :-
  ?o [ 1(?k) -> ?c ],
  ?o [ by -> ?o_by ],
  \symbol[ concat([By,?k,?o_by]) -> ?by
       ]@\basetype.
```

A.4 Rules in `core.flr`

Rules for deriving multi-level dependencies, propagating properties, aggregating properties, and for checking conformance.

```
// A5:
?o [ iof -> ?c1 ] :-
    ?o[ partOf->?o1, instanceOf->?c ],
  ?c1 [ level -> ?k, localTo -> ?o1 ],
  ?c [ level -> ?k ].

// A8:
?c1 [ iof -> ?m ] :-
  ?c1 [ localTo -> ?o, level -> ?k ],
  ?m [ localTo -> ?c, level -> ?k ],
  ?o [ iof -> ?c ].

// A9:
?c [ order -> ?i ] :-
  ?c.localTo [ order -> ?j ],
  ?i \is ?j + 1.

// A10:
?m1 [ partOfSchema -> ?m2 ] :-
  ?m1 [ localTo -> ?c, level -> ?k1 ],
  ?m2 [ localTo -> ?c, level -> ?k2 ],
  ?c1 [ localTo -> ?x, level -> ?k1 ],
  ?c2 [ localTo -> ?x, level -> ?k2 ],
  ?c1[ partOfSchemaT->?c, partOfSchema -> ?c2],
  ?c2 [ partOfSchemaT -> ?c ].

// A11:
?c1 [ partOfSchema -> ?c2 ] :-
   ?m1 [ partOfSchema -> ?m2 ],
  ?c1 [ iof -> ?m1] , ?c2 [ iof->?m2 ],
  ?c1 [ localTo -> ?x ], ?c2 [ localTo -> ?x].

 // --------- DIRECT SUBCLASS-OF ---------

// A12:
?c1 [ subclassOf -> ?c2 ] :-
    ?o1 [ partOf -> ?o2 ],
  ?c1 [ localTo -> ?o1, level -> ?k],
  ?c2 [ localTo -> ?o2, level -> ?k ].

// A13:
?m1 [ subclassOf -> ?m2 ] :-
  ?m1 [ localTo -> ?c1, level -> ?k ],
  ?m2 [ localTo -> ?c2, level -> ?k ],
  ?c1 [ subclassOf -> ?c2 ].

// ---------- MLT ------------

// A14:
?m [ partitions -> ?c ] :-
    ?o : DomainObject,
  ?c [ localTo -> ?o, level->?k ],
  ?c2 [ localTo -> ?o ],
  ?m [ localTo -> ?c2, level -> ?k].

// A15:
?m1 [ partitions -> ?m2 ] :-
  ?m1 [ localTo -> ?m, level -> ?k ],
  ?m2 [ localTo -> ?c, level -> ?k ],
  ?m [ partitions -> ?c ].

// A16:
?m1 [ subordinateTo -> ?m2 ] :-
   ?o : DomainObject,
  ?c1 [ localTo -> ?o, partOfSchema -> ?c2 ],
  ?m1 [ localTo -> ?c1, level -> ?k],
  ?m2 [ localTo -> ?c2, level -> ?k].

// A17:
?x [ subordinateTo -> ?y ] :-
   ?m [ subordinateTo -> ?m2 ],
  ?x [ localTo -> ?m, level -> ?k ],
  ?y [ localTo -> ?m2, level -> ?k].

 //FURTHER RULES NOT DISCUSSED IN DETAIL IN THE PAPER

// Classes can be adressed via methods of the object they
     are local to, e.g., Car.1(Model).1(Individual)
     returns CarIndividualClassByModel
?o [ 1(?k) -> ?c ] :-
  ?c [ localTo -> ?o, level -> ?k ].

?pow [ order -> ?i ] :-
  ?[ powertype -> ?pow, order -> ?j ],
  ?i \is ?j + 1.

// Derive transitive closure of partOfSchema
?c1 [ partOfSchemaT -> ?c3 ] :-
   ?c1 [ partOfSchema -> ?c2 ] ,
   ?c2 [ partOfSchemaT -> ?c3 ] .

?c1 [ partOfSchemaT -> ?c2] :-
  ?c1 [ partOfSchema -> ?c2 ].

// Go to root of localTo-nesting structure
?o [ localToDO -> ?o ] :-
  ?o : DomainObject .

?c [ localToDO -> ?do ] :-
  ?x [ localToDO -> ?do ],
  ?c [ localTo -> ?x ].

// subclassOfRT: recursive transitive closure of
     subclassOf
?c [ subclassOfRT -> ?c ] :-
   ?c.order > 0.

?c1 [ subclassOfRT -> ?c2 ] :-
   ?c1.subclassOfRT [ subclassOfRT -> ?c2].

?c1 [ subclassOfRT -> ?c2 ] :-
  ?c1 [ subclassOf -> ?c2 ] .

// Powertypes are included in subclassOfRT
?m1 [ subclassOfRT -> ?m2 ] :-
   ( ?c1 [ powertype -> ?m1 ] ;
      ?m1 [ partitions -> ?c1 ]),
   ?c1 [ subclassOfRT -> ?c2 ],
  ?c2 [ powertype -> ?m2 ].

// Class Membership
?o [ memberOf -> ?c ] :-
   ?o.iof [ subclassOfRT -> ?c ].
```

```
// Powertype Membership
?c [ memberOf -> ?pow ] :-
  ?c2 [ powertype -> ?pow ],
   ?c [ subclassOfRT -> ?c2 ] .

// propagateAsSharedValue induces shared property values
?c [ shared(?p) -> ?v ] :-
   ?m [ propagateAsSharedValue(?p) -> ?p2 ],
   ?c [ memberOf -> ?m , localToDO -> ?o ],
   ?o [ value(?p2) -> ?v ].

// propagateAsRange induces range refinements
?c [ range(?p) -> ?range ] :-
   ?m [ propagateAsRange(?p) -> c(?p2,?k) ],
   ?c [ memberOf -> ?m, localToDO -> ?o ],
   ?o [ value(?p2) -> ?o2 ],
  ?range [ localTo -> ?o2, level -> ?k ].

// A shared value becomes a property value with every
     member
?o [ value(?p) -> ?v ] :-
   ?c [ shared(?p) -> ?v ],
   ?o [ memberOf -> ?c ] .

// Calculate aggregated information
?c [ value(?p) -> ?v ] :-
   ?m [ aggregate(?p) -> ?exp ],
   ?c [ memberOf -> ?m ],
   eval(?c,?exp,?v).

eval( ?ctx, sum(?p), ?y ) :- ?y \is
   sum{?v|?_x [memberOf->?ctx,value(?p)->?v]}.

eval( ?ctx, avg(?p), ?y ) :- ?y \is
   avg{?v|?_x[memberOf->?ctx,value(?p)->?v]}.

eval( ?ctx, max(?p), ?y )  :-  ?y \is
   max{?v|?_x[memberOf->?ctx,value(?p)->?v]}.

eval( ?ctx, min(?p), ?y )  :-  ?y \is
   min{?v|?_x[memberOf->?ctx,value(?p)->?v]}.

 //------- CONFORMANCE CHECKS -------

// A class can only be modeled as part of a class if both
     are local to the same object
error(partOfIsLocal,[?c1,?c2]) :-
   ?c1[ partOfSchema -> ?c2 ],
   \naf exists(?o)^(
    ?c1 [ localTo -> ?o], ?c2 [ localTo -> ?o] ).

// A class can only be local to one object
error(localToIsFunctional,[?c]) :-
  ?c [ localTo -> ?o1 ],
  ?c [ localTo -> ?o2 ],
  ?o1 != ?o2.

// The schema of the part-whole hierarchy forms a forest
error(partOfFormsForest,[?c]) :-
   ?c[ partOfSchema -> {?c1,?c2} ],
   \naf (
      ?c1[partOfSchema->?c2];
      ?c2[partOfSchema->?c1];
      ?c1 = ?c2  ).

// A class cannot be associated with multiple levels
error(levelIsFunctional,[?c]) :-
  ?c [ level -> ?k1, level -> ?k2 ],
  ?k1 != ?k2.

// Level names are unique for the classes local to the
     same object
error(levelUniqueLocally,[?c,?c1,?o,?k]) :-
  {?c,?c1} [ localTo -> ?o, level -> ?k ],
   ?c != ?c1 .

// A class cannot have multiple orders
error(orderIsFunctional,[?o]) :-
   ?o [order -> {?i1, ?i2}], ?i1 != ?i2.

// An object cannot be direct part of multiple objects
error(partOfIsFunctional,[?o]) :-
   ?o [partOf -> {?o1, ?o2}], ?o1 != ?o2.

// An object can only be direct instance of one class
error(iofIsFunctional,[?o]) :-
   ?o [iof -> {?c1, ?c2}], ?c1 != ?c2.

// Classes that are modeled as part of another class
     cannot have direct instances
error(nonrootIofTopClass,[?o,?o1]) :-
   ?o [ partOf -> ?o1, iof->?c ],
  ?c [ partOfSchema -> ?_c2 ] .

// A refined range must be a subclassRT of the original
     range
error(propRangeSpec,[?c1,?c2,?p,?c3,?c4]) :-
   ?c1 [ subclassOfRT -> ?c2 ],
   ?c1 [ range(?p) -> ?c3  ],
   ?c2 [ range(?p) -> ?c4 ],
   \naf ( ?c3 [ subclassOfRT -> ?c4 ] ;
      ?c3 = ?c4 ; ?c3 :: ?c4 ).

// A refined metarange must be subclassRT of the original
     metarange
error(metarangeSpec,[?p,?m1,?m2,?m3,?m4]) :-
   ?m1 [ subclassOfRT -> ?m2 ],
   ?m1 [ metarange(?p) -> ?m3    ],
   ?m2 [ metarange(?p) -> ?m4 ],
   \naf ( ?m3 [ subclassOfRT -> ?m4 ] ).

// If a class defines a range for a property, then the
     property must be instantiated by all member objects
error(rangeNotInstantiated,[?o,?c,?p,?c2]) :-
   ?o [ memberOf -> ?c ],
   ?c [ range(?p) -> ?c2 ],
   \naf exists(?v)^(
      ?o[ value(?p) -> ?v ] ,
      ( (  \+ isnumber{?v}, ?v [ memberOf -> ?c2
           ] ) ; ?v : ?c2 )
   ).

// If a metaclass defines a metarange for a property, then
     all its member classes must define a range for that
     property
error(metarangeNotInst,[?c,?m,?p, ?m2]) :-
```

```
    ?c [ memberOf -> ?m ],
    ?m [ metarange(?p) -> ?m2 ],
    \naf exists(?c2)^( ?c [ range(?p) -> ?c2 ],
       ?c2 [ memberOf -> ?m2 ] ).

// Shared values must also conform to the property's range
error(sharedValueRange,[?c1,?c2,?p,?c3]) :-
    ?c1 [ subclassOfRT -> ?c2 ],
    ?c2 [ range(?p) -> ?c3 ],
    ?c1 [ shared(?p) -> ?v ],
    \naf ( ( \+ isnumber{?v}, ?v [ memberOf ->
         ?c3 ] ) ; ?v : ?c3 ) .

// Metarange-assertions must be consistent along
     subordination hierarchies
error(consistentMetarangeSubord,[?m1,?m2,?p]) :-
    ?m1 [ metarange(?p) -> ?m3,
            subordinateTo -> ?m2 ],
    ?m2 [ metarange(?p) -> ?m4 ],
    \naf ( ?m3 [ subordinateTo -> ?m4 ]  ) .

// A metarange-assertion must be consistent with property
     range given with partitioned class
error( propertyPartitioning, [?c,?m,?p] ) :-
    ?m [ partitions -> ?c,
          metarange(?p) -> ?m2 ],
    ?c[ range(?p) -> ?c2 ],
    \naf ( ?m2 [ partitions -> ?c2 ]  ) .
```

B Fully worked example in F-logic

In the example we use dot-notation to refer to local classes and powertypes. In this way the example facts can be executed with class names as well as with structured class IDs.

```
// --------- the SALES organisation hierarchy ------------
Sales : Hierarchy .

SalesDivision : Class [
  localTo -> Sales,
  level -> Division
].

SalesEmployee : Class [
  localTo -> Sales,
  level -> Employee,
  partOfSchema -> SalesDivision
].

SalesOutlet : Class [
  localTo -> Sales,
  level -> Outlet,
  partOfSchema -> SalesDivision
].

Vehicles : DomainObject [
  instanceOf -> SalesDivision
].

Electronics : DomainObject [
  instanceOf -> SalesDivision
].

MrWhite : DomainObject [
  partOf -> Vehicles,
  instanceOf -> SalesEmployee
].

MsJones : DomainObject [
  partOf -> Vehicles,
  instanceOf -> SalesEmployee
].

CarStoreVienna : DomainObject [
  partOf -> Vehicles,
   instanceOf -> SalesOutlet
].

BikesCarsLinz : DomainObject [
  partOf -> Vehicles,
  instanceOf -> SalesOutlet
].

MsBlack : DomainObject [
  partOf -> Electronics,
  instanceOf -> SalesEmployee
].

MsMuller : DomainObject [
  partOf -> Electronics,
  instanceOf -> SalesEmployee
].

EStoreVienna : DomainObject [
  partOf -> Electronics,
  instanceOf -> SalesOutlet
].
```

```
63
64  // --------- the PRODUCT hierarchy ------------
65  Product : Hierarchy .
66
67  ProductCategory : Class [
68    localTo -> Product ,
69    level -> Category,
70    range(vatRate) -> \number,
71    range(distributedBy) -> SalesDivision
72  ].
73
74  ProductModel : Class [
75    localTo -> Product ,
76    level -> Model,
77    partOfSchema -> ProductCategory,
78    range(productMgr) -> SalesEmployee,
79    range(listPrice) -> \number,
80    range(vatRate) -> \number
81  ].
82
83  ProductIndividual [
84    localTo -> Product ,
85    level -> Individual,
86    partOfSchema -> ProductModel,
87    range(soldPrice) -> \number,
88    range(soldAt) -> SalesOutlet,
89    range(soldTo) -> Customer,
90    range(vatRate) -> \number
91  ].
92
93  ProductCategory.1(Model) [
94    propagateAsRange(productMgr)
95      -> c( distributedBy, Employee ),
96    propagateAsSharedValue(vatRate) -> vatRate
97  ].
98
99  ProductCategory.1(Individual) [
100   propagateAsRange(soldAt)
101     -> c( distributedBy, Outlet ),
102   propagateAsSharedValue(vatRate)
103     -> vatRate
104 ].
105
106 ProductModel.powertype [
107   aggregate(avgListPrice) -> avg(listPrice)
108 ].
109
110 ProductIndividual.powertype [
111   aggregate(avgSoldPrice) -> avg(soldPrice)
112 ].
113
114 ProductModel.1(Individual).powertype [
115   aggregate(maxAvgSoldPricePerModel)
116     -> max(avgSoldPrice)
117 ].
118
119 ProductCategory.(Individual).powertype [
120   aggregate(maxAvgSoldPricePerCategory) ->
        max(avgSoldPrice)
121 ].
122
123 Car : DomainObject [
124   instanceOf -> ProductCategory ,
125   value(distributedBy) -> Vehicles ,
126   value(vatRate) -> 0.2
127 ] .
128
129 Car.1(Model) [
130   range(engineType) -> Engine.1(Type) ,
131   range(topSpeed) -> \number
132 ].
133
134 Car.1(Individual) [
135   range(engine) -> Engine.1(Individual),
136   range(maxSpeed) -> \number
137 ].
138
139 Car.1(Model).1(Individual) [
140   metarange(engine) ->
        Engine.1(Type).1(Individual) ,
141   propagateAsSharedValue(maxSpeed) -> topSpeed ,
142   propagateAsRange(engine) -> c( engineType,
        Individual )
143 ].
144
145 Phone : DomainObject [
146   instanceOf -> ProductCategory ,
147   value(distributedBy) -> Electronics ,
148   value(vatRate) -> 0.16
149 ] .
150
151 V50 : DomainObject [
152   instanceOf -> ProductModel ,
153   partOf -> Car ,
154
155   value(engineType) -> Vd4162T ,
156   value(productMgr) -> MrWhite ,
157   value(topSpeed) -> 190 ,
158   value(listPrice) -> 30000
159 ].
160
161 P911 : DomainObject [
162   instanceOf -> ProductModel ,
163   partOf -> Car ,
164
165   value(engineType) -> P3LTwinTurbo ,
166   value(productMgr) -> MsJones ,
167   value(topSpeed) -> 250 ,
168   value(listPrice) -> 120125
169 ].
170
171 IPhone6 : DomainObject [
172   instanceOf -> ProductModel ,
173   partOf -> Phone ,
174
175   value(productMgr) -> MsMuller ,
176   value(listPrice) -> 490
177 ].
178
179 MyCar : DomainObject [
180   instanceOf -> ProductIndividual ,
181   partOf -> V50 ,
```

```
    value(soldPrice) -> 26000 ,
    value(engine) -> Vd4162T_1 ,
    value(soldAt) -> BikesCarsLinz ,
    value(soldTo) -> Me
  ].

  AdasCar : DomainObject [
    instanceOf -> ProductIndividual ,
    partOf -> V50 ,

    value(soldPrice) -> 27000 ,
    value(engine) -> Vd4162T_2 ,
    value(soldAt) -> CarStoreVienna ,
    value(soldTo) -> Ada
  ].

  MiasCar : DomainObject [
    instanceOf -> ProductIndividual ,
    partOf -> P911 ,

    value(soldPrice) -> 119000 ,
    value(engine) -> P3TT_1 ,
    value(soldAt) -> BikesCarsLinz ,
    value(soldTo) -> Mia
  ].

  MiasPhone : DomainObject [
    instanceOf -> ProductIndividual ,
    partOf -> IPhone6 ,

    value(soldPrice) -> 439 ,
    value(soldAt) -> EStoreVienna ,
    value(soldTo) -> Mia
  ] .

  JoesPhone : DomainObject [
    instanceOf -> ProductIndividual ,
    partOf -> IPhone6 ,

    value(soldPrice) -> 459 ,
    value(soldAt) -> EStoreVienna ,
    value(soldTo) -> Joe
  ] .


  // --------- the ENGINE hierarchy ------------

  Engine : Hierarchy .

  EngineType : Class [
    localTo -> Engine ,
    level -> Type
  ] .

  EngineIndividual : Class [
    localTo -> Engine ,
    level -> Individual ,
    partOfSchema -> EngineType
  ] .

  Vd4162T : DomainObject [
    instanceOf -> EngineType
  ] .

  Vd4162T_1 : DomainObject [
    instanceOf -> EngineIndividual ,
    partOf -> Vd4162T
  ] .

  Vd4162T_2 : DomainObject [
    instanceOf -> EngineIndividual ,
    partOf -> Vd4162T
  ] .

  P3LTwinTurbo : DomainObject [
    instanceOf -> EngineType
  ] .

  P3TT_1 : DomainObject [
    instanceOf -> EngineIndividual ,
    partOf -> P3LTwinTurbo
  ] .

  // ----- objects not part of a hierarchy ------
  Me : Customer.
  Ada : Customer.
  Mia : Customer.
  Joe : Customer.
```

B.1 Running the example with structured class IDs `run_with_structures.flr`

```
#include "prepare.flr"
#include "class_creation_structures.flr"
#include "core.flr"
#include "example.flr"

config [ maxOrder -> 3 ].
```

B.2 Running the example with named class IDs `run_with_names.flr`

```
#include "prepare.flr"
#include "class_creation_names.flr"
#include "core.flr"
#include "example.flr"

config [ maxOrder -> 3 ].
```

B.3 Querying the multi-level model

Fig. 18 shows the result of querying all objects at all levels in the multi-level model.

```
?- ?[order->?order,level->?level], ?OBJECT [ order->?order, level->?level], (?OBJECT[ iof -> ?iof ]; (\naf
\exists(?_x)^(?OBJECT[ iof -> ?_x]), ?iof = nil)).
57 solution(s) in 0.172 seconds; elapsed time = 0.187
Yes
```

no	?order	?level	?OBJECT	?iof
1	0	Category	Car	ProductCategory
2	0	Category	Phone	ProductCategory
3	0	Division	Electronics	SalesDivision
4	0	Division	Vehicles	SalesDivision
5	0	Employee	MrWhite	VehiclesEmployee
6	0	Employee	MsBlack	ElectronicsEmployee
7	0	Employee	MsJones	VehiclesEmployee
8	0	Employee	MsMuller	ElectronicsEmployee
9	0	Individual	AdasCar	V50Individual
10	0	Individual	JoesPhone	IPhone6Individual
11	0	Individual	MiasCar	P911Individual
12	0	Individual	MiasPhone	IPhone6Individual
13	0	Individual	MyCar	V50Individual
14	0	Individual	P3TT_1	P3LTwinTurboIndividual
15	0	Individual	Vd4162T_1	Vd4162TIndividual
16	0	Individual	Vd4162T_2	Vd4162TIndividual
17	0	Model	IPhone6	PhoneModel
18	0	Model	P911	CarModel
19	0	Model	V50	CarModel
20	0	Outlet	BikesCarsLinz	VehiclesOutlet
21	0	Outlet	CarStoreVienna	VehiclesOutlet
22	0	Outlet	EStoreVienna	ElectronicsOutlet
23	0	Root	Engine	nil
24	0	Root	Product	nil
25	0	Root	Sales	nil
26	0	Type	P3LTwinTurbo	EngineType
27	0	Type	Vd4162T	EngineType
28	1	Category	ProductCategory	nil
29	1	Division	SalesDivision	nil
30	1	Employee	ElectronicsEmployee	SalesEmployeeClassByDivision
31	1	Employee	SalesEmployee	nil
32	1	Employee	VehiclesEmployee	SalesEmployeeClassByDivision
33	1	Individual	CarIndividual	ProductIndividualClassByCategory
34	1	Individual	EngineIndividual	nil
35	1	Individual	IPhone6Individual	PhoneIndividualClassByModel
36	1	Individual	P3LTwinTurboIndividual	EngineIndividualClassByType
37	1	Individual	P911Individual	CarIndividualClassByModel
38	1	Individual	PhoneIndividual	ProductIndividualClassByCategory
39	1	Individual	ProductIndividual	nil
40	1	Individual	V50Individual	CarIndividualClassByModel
41	1	Individual	Vd4162TIndividual	EngineIndividualClassByType
42	1	Model	CarModel	ProductModelClassByCategory
43	1	Model	PhoneModel	ProductModelClassByCategory
44	1	Model	ProductModel	nil
45	1	Outlet	ElectronicsOutlet	SalesOutletClassByDivision
46	1	Outlet	SalesOutlet	nil
47	1	Outlet	VehiclesOutlet	SalesOutletClassByDivision
48	1	Type	EngineType	nil
49	2	Employee	SalesEmployeeClassByDivision	nil
50	2	Individual	CarIndividualClassByModel	ProductIndividualMetaclassByModelByCategory
51	2	Individual	EngineIndividualClassByType	nil
52	2	Individual	PhoneIndividualClassByModel	ProductIndividualMetaclassByModelByCategory
53	2	Individual	ProductIndividualClassByCategory	nil
54	2	Individual	ProductIndividualClassByModel	nil
55	2	Model	ProductModelClassByCategory	nil
56	2	Outlet	SalesOutletClassByDivision	nil
57	3	Individual	ProductIndividualMetaclassByModelByCategory	nil

Fig. 18 Result of querying all objects in the multi-level model, ordered by their order and level in the domain object hierarchy, together with their class (?iof) if present

References

1. Atkinson, C.: Meta-modeling for distributed object environments. In: Proceedings of the 1st International Enterprise Distributed Object Computing Conference. IEEE Computer Society (1997)
2. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. In: Gogolla, M., Kobryn, C. (eds.) Proceedings of the 4th International Conference on the UML 2001, Toronto, Canada, LNCS, vol. 2185, pp. 19–33. Springer (2001)
3. Atkinson, C., Kühne, T.: Model-driven development: a metamodeling foundation. IEEE Softw. **20**(5), 36–41 (2003)
4. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Softw. Syst. Model. **7**(3), 345–359 (2008). https://doi.org/10.1007/s10270-007-0061-0
5. Atkinson, C., Kühne, T.: In defence of deep modelling. Inf. Softw. Technol. **64**, 36–51 (2015). https://doi.org/10.1016/j.infsof.2015.03.010
6. Balaban, M., Khitron, I., Kifer, M., Maraee, A.: Formal executable theory of multilevel modeling. In: Krogstie, J., Reijers, H.A. (eds.) Advanced Information Systems Engineering—30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11–15, 2018, Proceedings, Lecture Notes in Computer Science, vol. 10816, pp. 391–406. Springer (2018). https://doi.org/10.1007/978-3-319-91563-0_24
7. Cardelli, L.: Structural subtyping and the notion of power type. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 70–79 (1988)
8. de Carvalho, V.A., Almeida, J.P.A.: Toward a well-founded theory for multi-level conceptual modeling. Softw. Syst. Model. **17**(1), 205–231 (2018). https://doi.org/10.1007/s10270-016-0538-9
9. Carvalho, V.A., Almeida, J.P.A., Fonseca, C.M., Guizzardi, G.: Extending the foundations of ontology-based conceptual modeling with a multi-level theory. In: Johannesson, P., Lee, M., Liddle, S.W., Opdahl, A.L., López, O.P. (eds.) ER 2015, LNCS, vol. 9381, pp. 119–133. Springer (2015)
10. Carvalho, V.A., Almeida, J.P.A., Fonseca, C.M., Guizzardi, G.: Multi-level ontology-based conceptual modeling. Data Knowl. Eng. **109**, 3–24 (2017)
11. Eriksson, O., Henderson-Sellers, B., Ågerfalk, P.J.: Ontological and linguistic metamodelling revisited: a language use approach. Inf. Soft. Technol. **55**(12), 2099–2124 (2013)
12. Fine, K.: Towards a theory of part. J. Philos. **107**(11), 559–589 (2010)
13. Gangemi, A., Guarino, N., Masolo, C., Oltramari, A.: Sweetening WORDNET with DOLCE. AI Mag. **24**(3), 13–24 (2003). https://doi.org/10.1609/aimag.v24i3.1715
14. Gómez, C., Olivé, A.: Evolving derived entity types in conceptual schemas in the UML. In: Konstantas, D., Léonard, M., Pigneur, Y., Patel, S. (eds.) Object-Oriented Information Systems, 9th International Conference, OOIS 2003, Geneva, Switzerland, September 2–5, 2003, Proceedings, Lecture Notes in Computer Science, vol. 2817, pp. 33–45. Springer (2003). https://doi.org/10.1007/978-3-540-45242-3_5
15. Gonzalez-Perez, C., Henderson-Sellers, B.: A powertype-based metamodelling framework. Softw. Syst. Model. **5**(1), 72–90 (2006)
16. Guizzardi, G.: Ontological foundations for structural conceptual models. Ph.D. Thesis, University of Twente, The Netherlands (2005)
17. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S.: ConceptBase—a deductive object base for meta data management. J. Intell. Inf. Syst. **4**(2), 167–192 (1995)
18. Jeusfeld, M.A.: Complete list of O-Telos axioms (2005). Online http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/d1228997/O-Telos-Axioms.pdf
19. Jeusfeld, M.A., Almeida, J.P.A., Carvalho, V.A., Fonseca, C.M., Neumayr, B.: Deductive reconstruction of mlt* for multi-level modeling. In: Guerra, E., Iovino, L. (eds.) MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18–23 October, 2020, Companion Proceedings, pp. 83:1–83:10. ACM (2020). https://doi.org/10.1145/3417990.3421410
20. Kappel, G., Schrefl, M.: Local referential integrity. In: International Conference on Conceptual Modeling, pp. 41–61. Springer (1992)
21. Klas, W., Schrefl, M.: Metaclasses and Their Application—Data Model Tailoring and Database Integration. Springer (1995)
22. Kuehne, T.: A story of levels. In: MODELS Workshops, pp. 673–682 (2018)
23. de Lara, J., Guerra, E., Cobos, R., Moreno-Llorena, J.: Extending deep meta-modelling for practical model-driven engineering. Comput. J. **57**(1), 36–58 (2014)
24. de Lara, J., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Trans. Softw. Eng. Methodol. **24**(2), 12:1-12:46 (2014)
25. Macías, F., Wolter, U., Rutle, A., Durán, F., Rodriguez-Echeverria, R.: Multilevel coupled model transformations for precise and reusable definition of model behaviour. J. Log. Algebr. Methods Progr. **106**, 167–195 (2019)
26. Mylopoulos, J.: Conceptual modelling and telos. In: Loucopoulos, P., Zicari, R. (eds.) Conceptual Modelling, Databases, and CASE: An Integrated View of Information System Development, pp. 49–68. Wiley, USA (1992)
27. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: representing knowledge about information systems. ACM Trans. Inf. Syst. **8**(4), 325–362 (1990)
28. Neumayr, B., Grün, K., Schrefl, M.: Multi-level domain modeling with M-objects and M-relationships. In: Link, S., Kirchberg, M. (eds.) APCCM, CRPIT, vol. 96, pp. 107–116. ACS, Wellington (2009)
29. Neumayr, B., Schrefl, M., Thalheim, B.: Modeling techniques for multi-level abstraction. In: Kaschek, R., Delcambre, L.M.L. (eds.) The Evolution of Conceptual Modeling, LNCS, vol. 6520, pp. 68–92. Springer (2008)
30. Neumayr, B., Schuetz, C.G., Jeusfeld, M.A., Schrefl, M.: Dual deep modeling: multi-level modeling with dual potencies and its formalization in f-logic. Softw. Syst. Model. **17**(1), 233–268 (2018). https://doi.org/10.1007/s10270-016-0519-z
31. Odell, J.: Power types. JOOP **7**(2), 8–12 (1994)
32. Partridge, C., de Cesare, S., Mitchell, A., Gailly, F., Khan, M.: Developing an ontological sandbox: Investigating multi-level modelling's possible metaphysical structures. In: L.B. et al. (ed.) Proceedings of MODELS 2017 Satellite Event, CEUR Workshop Proceedings, vol. 2019, pp. 226–234. CEUR-WS.org (2017). http://ceur-ws.org/Vol-2019/multi_3.pdf
33. Pirotte, A., Zimányi, E., Massart, D., Yakusheva, T.: Materialization: a powerful and ubiquitous abstraction pattern. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) VLDB, pp. 630–641. Morgan Kaufmann (1994)
34. Selway, M., Stumptner, M., Mayer, W., Jordan, A., Grossmann, G., Schrefl, M.: A conceptual framework for large-scale ecosystem interoperability. In: Johannesson, P., Lee, M., Liddle, S.W., Opdahl, A.L., López O.P. (eds.) ER 2015, LNCS, vol. 9381, pp. 287–301. Springer (2015)

Bernd Neumayr is senior researcher and lecturer at Johannes Kepler University (JKU) Linz, Austria. He received his doctorate degree from JKU in 2010. In 2015/2016, he was a visiting researcher at the Department of Computer Science of the University of Oxford. His main research interests include intelligent information systems for air traffic management and multi-level conceptual modeling.

Michael Schrefl received his Dipl.-Ing. degree and his Doctorate from Vienna University of Technology, Vienna, Austria, in 1983 and 1988, respectively. During 1983–1984, he studied at Vanderbilt University, USA, as a Fulbright scholar. From 1985 to 1992, he was with Vienna University of Technology. During 1987–1988, he was on leave at GMD IPSI, Darmstadt, where he worked on the integration of heterogeneous databases. He was appointed Professor of Information Systems at Johannes Kepler University of Linz, Austria, in 1992, and Professor in Computer and Information Science at University of South Australia in 1998. He currently leads the Department of Business Informatics-Data and Knowledge Engineering at Johannes Kepler University of Linz, with projects in data warehousing, workflow management, and web engineering.