Streamlining Conceptual Modeling

Hermann Bense

bense.com GmbH, Schwarze-Brüder-Str. 1, 44137 Dortmund, Germany,
hb@bense.com, https://orcid.org/0000-0002-2562-224X

Abstract. Research into knowledge and expert system engineering has seen numerous attempts to streamline conceptual modeling, especially in the field of so-called *Multi-Level Modeling* (MLM). In this paper, we will refine and augment the approaches made there, so that these can also be used for conceptual modeling in general. To this end, we introduce a formal methodology for the definition of attributes (data properties) and relational relationship types (object properties). Rules are then defined as instantiation regulations in this connection. As a new methodology, we introduce the so-called *Value Assignment Propagation* (VAP), and then show how this streamlines conceptual modeling, how layer-mistakes can be avoided and how previously unfulfilled requirements for MLM can be satisfied.

Keywords: Knowledge Engineering, Conceptual Modeling (CM), Multi-Level Modeling (MLM), Data Properties (DP), Value Assignment Propagation (VAP), Filters, Inheritance, (Deep) Instantiation, Powertypes

1 Introduction

The past two decades have been a number of seminal publications and books about ontology engineering, yet, we are unable to arrive at a mathematical formalism for instantiation. Intuition is the predominant aspect in identifying what data properties can be invoked via inheritance, but a precise definition of which sets of universals and particulars are affected, is nowhere to be found; in particular, the question of how inheritance affects object properties (relationship types) with regard to instantiation is left largely unanswered.

The way in which the terms *inheritance* and *instantiation* are applied by ontologists is often features an intuitive type of understanding, which is based on the methods of object-oriented programming (OOP), object-oriented analysis (OOA) and object-oriented design (OOD). *The Unified Modeling Language* (UML) and UML diagrams are utilized to visualize conceptual models and to specify the logic of applications. Ontologistcs also rely on modeling languages such as the Web Ontology Language

(OWL), which is based on *Description Logic* (DL). The concepts of inheritance and instantiation also play a key role in relational and NO-SQL database systems. When attempting to actually break the modeling down into its implementation levels, a mismatch often occurs because the methodology and the power of the various representations vary from each other. By applying mathematical set theory, we will extend and generalize the formalisms for describing inheritance and instantiation and we will have to distance ourselves from known methods, where necessary.

The rest of the paper is organized as follows: In the Related Work section (s. 2), we discuss what approaches already exist for streamlining conceptual modeling and what disadvantages they entail. The Methodology section (s. 3) describes the naming conventions on which our methodology are based and the mathematical foundations for the formalizing inheritance and instantiation. Using concrete modeling examples, the application of the methodology is discussed in the Application section (s. 4). In the Evaluation section (s. 5), we evaluate the extent to which our methodology contributes to fulfilling the criteria for Multi-Level Modeling (MLM), and to what extent our approach contributes to streamlining conceptual modeling in general, and how metadata can be managed more systematically; we also document the savings potential of our modeling. Finally, in the Conclusion section (s. 6), we summarize the results and describe the outlook fore future research activities.

2 Related Work

The article by Humm et al. [1] differentiates between lightweight and heavyweight ontologies and associates developer-oriented communities with the first, and academic communities with the latter. Our primary motivation is to introduce a methodology for conceptual modeling that has no limitations to the power of modeling, but avoids the ballast and complexity of other approaches. That is, our approach aims to support the development of both lightweight and heavyweight ontologies. As a typical layer mistake in *Meta-Object Facility* (MOF)[1] we find that `Class` appears on both levels 2 and 3, which we believe to be a flaw. The article by Carvalho and Almeida [2] describes how a powertype `MobilePhoneModel` can be modelled using *regularity attributes*. But the classes the `MobilePhone` and `MobilePhoneModel` cannot be merged with regularity attributes, since each particular `phone` would then also have a `launchDate`. Another current subject of discussion in ontology modeling and in the Semantic Web is

[1] https://en.wikipedia.org/wiki/Meta-Object_Facility

the question of how we can model *Labeled Property Graphs* (LPG). Suggested solutions come from the NO-SQL community, particularly from neo4j[2]. So far, however, these have not been systematically incorporated into modeling languages such as RDF and OWL. Within the Dublin Core Metadata Initiative[3], metadata is defined in the *Dublin Core Metadata Element Set*. Are data properties like `Title`, `Publisher` and `Description` really always metadata, or do they belong more to the application domain? There needs to be a clarification on how to achieve flexibility in modeling so that such attributes can be used either as meta attributes or attributes of the application domain, or in both ways, if necessary. An article [3] by Atkinson and Kühne argues that the shallow instantiation approach cannot solve the issues surrounding *multiple classification* and *replication of concepts* problems. The authors add the concept of *potency* to each model element. Potency defines the depth to which a model element can be instantiated. In each instantiation step, the potency is reduced by 1. However, this requires the concepts to be modeled such that they can be divided into logical levels, each of which must be assigned a meaning. This harbors the potential for layer mistakes, especially in ontologies that are undergoing frequent changes during development. The subdivisions of base classes are also often performed using terms such as `category, family, group, order, subordinate`, etc. For example, in a paper by Neumayr et al. [4] on page 9, in Figure 3, the basic classes `Book` and `Car` are modeled as subclasses of the `ProductCategory` and assigned a classification level of 3. To model that `marketLauch` has to be instantiated at brand level and `mileage` has to be instantiated by objects at level physical entity, the authors claim it is necessary to use explicit constraints. Since the authors do not specifiy the constraints, the question of how this can be achieved is left open. We also see the potential for layer mistakes in taking this approach. A paper by Guizzardi et al. [5] also discusses *deep instantiation*. The classes `BirdSpecies` and the class `Bird` are modelled in two parallel class hierarchies. In this we identifiy a modeling deficiency whereby the characteristics of all other species would have to be modeled redundantly as in `BirdSpecies`, or would need to be shifted to a higher class `Species`. The authors' modeling of roles for `Gender` and `ThreatPhase` does not answer how one of the values for the enumeration types should be associated with a specific particular. We make the assumption that it should be a 1:1 connection via an object property. Nevertheless, we consider the graph rewriting rules on page 6 of [6] to represent an important

[2] https://neo4j.com/
[3] https://www.w3.org/2001/09/rdfprimer/rdf-primer-20021108.html#dublincore

contribution in reducing the complexity of ontology models. The approach of Bense and Humm [7] proposes satisfying the instance and type character via two different inheritance mechanisms of properties. These are `BroderInheritedProperty` based on `skos:broader` and `TypeInheritedProperty` based on `rdfs:class`. The case that is not covered by this is if we want to model that all `polar bears` have a `white fur`.

Other proposals made in the area of *Multi-Level Modeling* (MLM) ([8], [9], [10], [11]) describe approaches with which conceptual modeling is not only enriched but also streamlined. What the cited approaches have in common, however, is that they require additional complex or even overly-complex annotations and/or the extension of query languages, meaning they are no longer compatible with RDF/RDFS/OWL and query languages such as SQL/SPARQL. In addition, not all of the requirements previously placed on on MLM in [5], [9], and [12] could be met. So, major problems in conceptual modelling result from the non-availability of suited uniform methods or the inadequate application of modeling methods. In addition to other problems, this causes *layer mistakes*, which penetrate even into the basic methods of software engineering.

The key research question we answer is: What are the limitations in ontology engineering when it comes to creating leaner, less redundant, better comprehensive and more accurate models, and what methods can be used in overcoming the limitations discussed above. This raises other additional questions: What naming conventions can be used to reduce ambiguities? What patterns can be specified to define properties and what are the permissible instantiations that result from this? What influence does this have on the modeling style of the ontologists? How can existing ontologies benefit from this, and what advantages emerge here for maintaining accuracy and consistency in the evolution of ontologies.

3 Methodology

Before answering questions, the methodology used here has to be introduced. Basic methodologies for naming conventions in ontological engineering were described in [13]. Figure 1 is a so-called *ontograph* visualization of an example of a vehicle ontology that makes use of these naming conventions. The open source library graphviz[4] ist used to create ontographs. The quick switching between ontology modeling and visualization supports rapid prototyping, thus supporting error detection and

[4] http://www.graphviz.org

correction. The example in Figure 1 also illustrates the principle of (multiple) inheritance, as well as depicting the instantiation of object properties. The following sections will explain in greater detail the naming conventions used, the different types of property definitions, and their instantiations.

But first of all, we need to introduce a set of basic definitions, naming conventions, axioms and rules for conceptual modeling. Our aim is to manage with as few concepts as possible. These should be understood as recommendations and they may be adapted or implemented in alternative ways. For example, the prefixes used in the naming conventions encode the type of the concepts within their names. Of course, by explicitly specifying further meta information, each represented concept can alternatively be classified without using prefixes. The core concepts we define here include *Knowledge Graph* (KG), *Knowledge Subject* (KS), *Atomic Data Type* (ADT), V*alue Set* (VS), *Value Set Range* (VSR) and *Particulars Layers* (PL) as well as *Universals Layer* (UL). The mathematical concepts *Principal Filter* and *Principal Ideal* allow for a compact definition of hierarchies of universals. The aforementioned concepts consequently form the basis for formally defining *Data Property Definitions* (DPD) and *Object Property Definitions* (OPD). The instantiation rules then determine how DPDs and OPDs may be applied. The instantiation of data properties is referred to as *Data Property Instantiation* (DPI), and that of object properties is referred to as *Object Property Instantiation* (OPI). *Relator Instantiation* is a special type of OPI for modeling n-ary relationships. The central approach taken by this paper is called *Value Assignment Propagation* (VAP) and consists of introducing a new methodology for the different application of the data property instantiation (DPI) based on two different types of the data properties, namely the *Propagation Data Property* (PDP) and the *Meta Data Property* (MDP).

Definitions: The key concepts in ontology engineering are classes, attributes, relationship types, particulars, relators, processes and functions. In line with language usage in the Semantic Web, we also refer to attributes as *Data Properties* (DP) and relationship types as *Object Properties* (OP). It is therefore consistently true that a property p is either a DP or an OP. Classes, object properties and processes are also gathered under the general term *universal*. The most important concepts in ontology engineering are classes, attributes, relationship types, particulars, relators, processes and functions. According to the language usage in the Semantic Web, we also refer to attributes as *Data Properties* (DP) and relationship types as *Object Properties* (OP). It applies consistently that a property

p is either a DP or an OP. Classes, object properties and processes are also collected under the general term *universal*. Particulars are instances of universals which are connect by the object property `»iof` (is instance of) with a universal, e.g. `(>Toms_Lambo, »iof, ^Lamborghini_Miura)`. Besides through duplication particulars cannot be instantiated (s. Axiom A1 in [8], Page 8). Thus, the assertion `(x, »iof, >Toms_Lambo)` is not allowed.

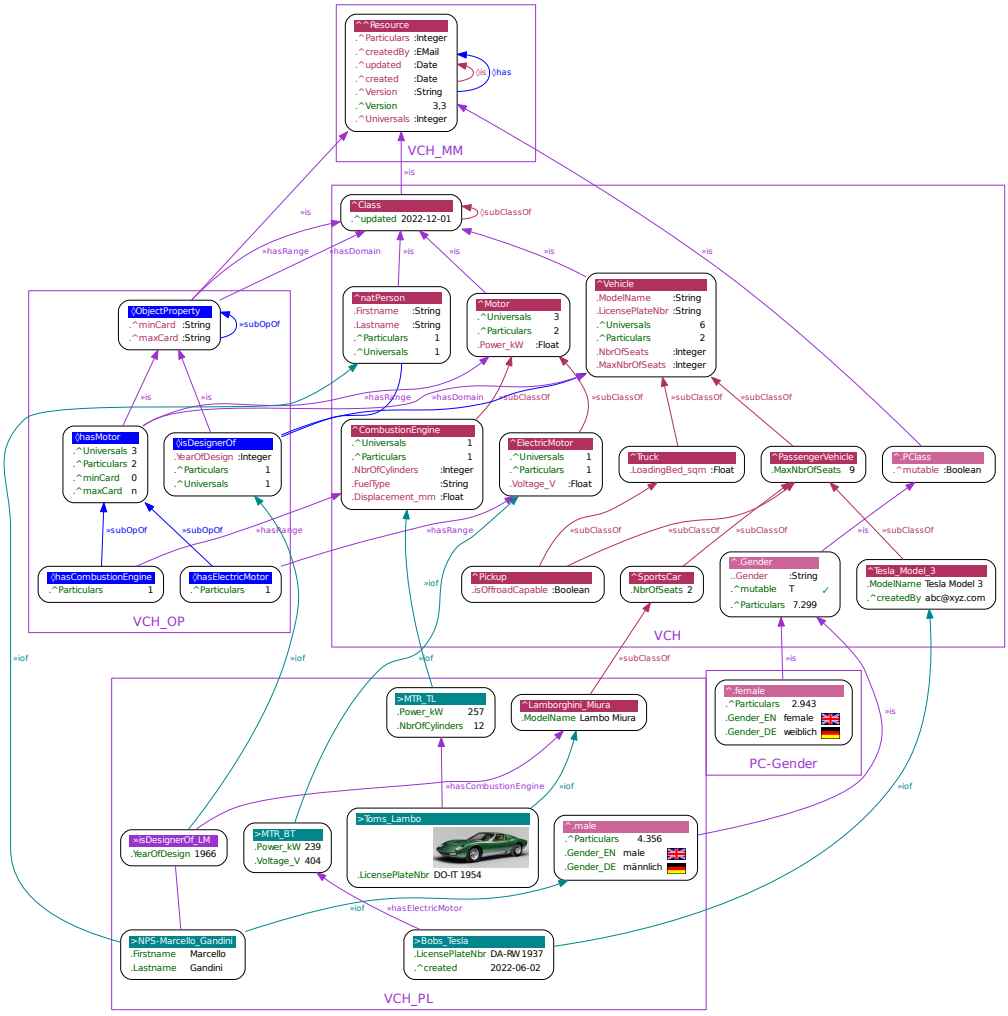

Fig. 1: Vehicle Ontology

Name Sets: The name of an ontological concept always has the index of the designating set as its prefix: $N_x = \{xn \mid n \text{ is a string}\}$. To enable the names of ontological concepts to be unambiguously designated, we intro-

duce the following sets: **N**=Strings with the subsets **N**$_{\hat{}}$=Class Names, **N**$_{.}$=Data Property Names, **N**$_{\Diamond}$=Object Property Names, **N**$_{>}$=Particular Names, **N**$_{»}$=Relator Names, **N**$_{\circ}$=Process Names, **N**$_{\sim}$=Function Names. The names of universals is the set $N_U = N_{\hat{}} \cup N_{\Diamond} \cup N_{\circ}$. Later, we will show how the prefixes of data properties, in particular, are used to define instantiation rules. Similar naming conventions for so-called regularity attributes are introduced in [2], e.g. the prefix `instances` in `instancesScreenSize` indicates that an attribute of the type class `MobilePhoneModel` correlates with the corresponding attribute of the base class `MobilePhone`.

Knowledge Graphs: An ontology is represented by a *Knowledge Graph* (KG). The smallest syntactic storage unit is a (RDF) triple[5] in the form `(s, p, o)`. A *Knowledge Graph* (KG) is defined as KG $\subseteq$ N x N x N. With s, p, o $\in$ N, a *Knowledge Atom* (KA) is a triple `(s, p, o)` $\in$ `KG`. A *Knowledge Subject* (KS) is the set of triples with the same subject x is denoted by Σ_x:

Df 31 (Knowledge Subject)**.** $\Sigma_x = \{(s, p, o) \in KG \mid s = x\}$.

A knowledge subject Σ_x is represented as a node in the graphical representation of a knowledge graph (OntoGraph) and has the node name x. According to our understanding, a knowledge subject also corresponds to an instance / singleton that is unique, regardless of whether it is a particular or a universal. Even if all pairs (p,o) of values of two knowledge subjects Σ_x and Σ_y with $x \neq y$ are identical, they are different objects simply because their names x and y are different. The physical representation of a knowledge graph is called a *Knowledge Repository* (KR). A KR can be a file, a table, or a database.

Atomic Data Types: Data properties are defined by exactly one so-called *Atomic Data Type* (ADT). An ADT name is an element of the set `ADT = {:String, :Integer, :Decimal, :Float, :Double, :Boolean, :Duration, :DateTime, :Date, :Time, :anyURI, ...}`. To simplify matters here, we only use a subset of the datatypes defined in OWL[6], and XML[7], and we also use a different notation. Instead of `DataPropertyAssertion(a:NbrOfCylinders a:>MTR_TL "12"^^ xsd:integer)`, we would make assignments in the more readily editable and readable form `(>MTR_TL, .NbrOfCylinders, 12)`. Nor do we encode the type of the attribute value at the attribute value itself, but rather within the definition of the data property with `(^CombustionEngine, .NbrOfCylinders, :Integer)` instead. We consider it a major disadvantage that, according to OWL conventions, the atomic data type always must

[5] https://en.wikipedia.org/wiki/Semantic_triple
[6] https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/#Datatype_Maps
[7] https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/#anyAtomicType

be specified with the value itself, as this leads to unnecessarily high additional storage overheads and needlessly complicates queries and makes comparisons difficult.

Value Sets: The set of all possible values of a data property p is defined by the *Value Set Range* (VSR). The set of all values actually applied in the ontology for a data property form the *Value Set* VS(p) ⊆ VSR(p).

Df 32 (Value Set)**.** VS(p) = { v | (s, p, v) ∈ KG}. Examples:

- `VSR(:Natural)` = ℕ, `VSR(:Integer)` = ℤ, `VSR(:Float)` = ℝ
- `VSR(:Boolean)` = {0, 1} ⋁ {T, F}; `VS(.warmblooded)` = {T, F}
- `VSR(.Gender)` = :String; `VS(.Gender)` = {male, female}
- `VSR(.WaveLengthOfVisibleLight_nm)` = {wl | wl ∈ [400, 780]}

Layers: We differentiate between two ontology layers, the *Universals Layer* (UL) and the *Particulars Layer* (PL). The PL contains all particulars which are connected via the »`iof` object property to an universal and all object property instantiations (OPI) between any two particulars. Universals are organized into hierarchies by the following sets of *semantic relations* (SR).

- `SRU` = {◊`is`}; `SRM` = {◊`has`};
- `SRR` = {◊`subClassOf`, ◊`subOpOf`, ◊`subDpOf`, ◊`subProcessOf`}
- `SRO` = {◊`hasSubClass`, ◊`hasSubOp`, ◊`hasSubDp`, ◊`hasSubProcess`}

The object properties of the SSR set express hype**R**nomy, those of the SRO set hyp**O**nomy and the respective inverse relationship results from the naming convention: ◊$_{\text{subXOf}}$ = ◊$_{\text{hasSubX}}^{-1}$. The UL contains all universals including all data property definitions (DPD), all object property definitions (OPD) and all hierarchy relationships defined through an object property `op` ∈ `SR` = `SRR` ∪ `SRO`. The number of levels in a hierarchy is not restricted. Cycles are not permitted in hierarchies.

Filters: Let ≤ be a non-strict order relation, while the pair (M, ≤) is called a non-strictly ordered set. A filter or *order filter* (OF) is a special subset of a partially ordered set (poset). We use filters to construct parent and child hierarchies of universals. With `x, y` ∈ `N`$_{\text{U}}$ and ◊srr ∈ SRR or ◊sro ∈ SRO we define:

- `x`≤`y` ⇔ (`(x,` ◊`ssr, y)` ∈ `KG` or `(y,` ◊`ssr`$^{-1}$`, x)` ∈ `KG)`
- `x`≥`y` ⇔ (`(x,` ◊`sro, y)` ∈ `KG` or `(y,` ◊`sro`$^{-1}$`, x)` ∈ `KG)`

Df 33 (principal filter)**.** ∵a = {x∈M| a≤x} is the principal filter of a.

Df 34 (principal ideal)**.** ∴a = {z∈M| z≤a} is the principal ideal of a.

Inheritance: We use the term inheritance essentially as it is applied in programming. Properties are inherited across a hierarchy of universals. As the root element of all such inheritance hierarchies, we introduce the `^^Resource` element. The elements of a class hierarchy are connected by the transitive object property ◊`subClassOf`. A class is by definition a subclass of itself (as the subset may be the complete set)[8]. For example, the triples `(^ElectricMotor, »subClassOf, ^Motor)`, `(^CombustionEngine, »subClassOf, ^Motor)` and `(^Motor, »subClassOf, ^Class)` define a so-called class hierarchy (s. Fig. 1). The data property `.Power_kW` is therefore inherited from the `^Motor` class to all of its subclasses. This means that any particular motor can be assigned a value for the `.Power_kW` data property.

Inheritance Hierarchies: Be `u` ∈ N_U the name of an universal. The principal filter ⋱u is the set of all super universals of u including u, the principal ideal ⋰u is the set of all sub universals of u including u, e.g.:

- ⋱`^Pickup = {^Pickup, ^Truck, ^PassengerVehicle, ^Vehicle}`
- ⋰`^Motor= {^Motor, ^CombustionEngine, ^ElectricMotor}`
- ⋱◊`hasCombustionEngine = {`◊`hasCombustionEngine,` ◊`hasMotor}`
- ⋰◊`hasMotor = {`◊`hasMotor,` ◊`hasElectricMotor,` ◊`hasCombustionEngine}`

Property Definitions: We distinguish between data and object property definitions. For both, we provide a compact and a traditional definition pattern. The two triple definition pattern, which uses the object properties `»hasDomain` and `»hasRange`, can be always transformed into the compact one triple definition pattern.

Data Property Definitions: Internal properties of instances are referred to as data properties (DP) according to the convention in the Semantic Web / OWL / RDF. These include characteristics such as `.Color`, `.Length`, `.Weight`, `.Age`, etc., which establish the corresponding quality dimensions. Traditionally a *Data Property Definition* (DPD) is built using two triples e.g. `(.LicensePlateNbr, »hasDomain, ^Vehicle)` and `(.LicensePlateNbr, »hasRange, :String)`. The short form of a DPD determines the associated type for the class, e.g. `(^Vehicle, .LicensePlateNbr, :String)` and an instantiation like `(>Toms_Lambo, .LicensePlateNbr, DO-IT 1954)`. Be a ∈ $N_.$ the name of a data property, `:adt` ∈ `ADT` an Atomic Data Type, and u ∈ N_U the name of a universal. Both versions of the DP definitions are equivalent according to the following rule: `(u, a, :adt)` ⇔ `(a, »hasDomain, u)` ∧ `(a, »hasRange, :adt)`. So, data properties (DP) connect instances to values from the value set VS(DP). *Data Properties* (DP) are defined using *Atomic Data Types* (ADT) with an associated Value Set (VS). We distinguish between two fundamentally different types of data properties with

[8] https://www.w3.org/TR/owl-ref/#subClassOf-def

their *Data Property Definition Patterns* (DPDP) defined below. DPDPs are inherited down the universal hierarchy, in which they are defined. Be `.dp` the name of a *Propagation Data Property* (PDP) and `.^dp` the name of a *Meta Data Property* (MDP).

- A *Propagation Data Property Definition* (PDPD) has the form `(u, .dp, :adt)`, for example, `(^Motor, .Power_kW, :Float)`.
- A *Meta Data Property Definition* (MDPD) has the form `(u, .^dp, :adt)`, for example, `(^Car, .^YearOfDesign, :String)`.

Object Property Definitions: External properties of instances are referred to as *Object Properties* (OP) according to the convention in the Semantic Web / OWL / RDF. Object properties connect instances with another instance or with itself. The traditional definition for an object property e.g. `◊hasMotor` is defined using the two triples `(◊hasMotor, »hasDomain, ^Car)` and `(◊hasMotor, »hasRange, ^Motor)`. The short form of an *Object Property Definition* is `(^Car, ◊hasMotor, ^Motor)`. As an example from the vehicle ontology we have `(>Toms_Lambo, ◊hasCombustionEngine, >MTR_TL)` where `◊hasCombustionEngine` is a subobject property (`◊subOpOf`) of `◊hasMotor`. This modeling is an example of how it is possible to define relationship types between instances and their specialization on different abstraction levels according to the requirement set out in [12], page 3. Be `◊op ∈ N◊` the name of an object property, and `c, d ∈ N^` the name of classes. Both versions of the OP definitions are equivalent according to the following rule: `(c, ◊op, d)` ⇔ `(◊op, »hasDomain, c)` ∧ `(◊op, »hasRange, d)`.

Instantiation: How can we formally define *instantiation*? We essentially understand instantiation to be the creation of a new knowledge subject corresponding to a node in the knowledge graph, whose identifier `x` ≠ `y` for all previously existing instances of y. During the creation process, all possible value assignments should be preceded by the typing of the instance either as a particular of a class with `(x, »iof, U)` or as a subclass of another class with `(x, »subClassOf, y)`. Any further value assignments can then be made to the instance, regardless of whether they are data properties or object properties. Be `^C, ^D ∈ N^` the names of classes and `∴^C` and `∴^D` their superclass hierarches and `.∴^C` and `.∴^D` their subclass hierarches and `SOP(∴u)` = ∪ `{sop(c)` | `c ∈ ∴u}`. Be `>P, >P1, >P2 ∈ N>` the name of particulars and `>P1 ∈ SOP(∴^C), >P, >P2 ∈ SOP(.∴^D)`. Be `r ∈ N◊` the name of an object property and `∴r` the set of names of the super object properties of r. Be `DP ∈ N.` a data property name and `v ∈ VS(DP)` a value from the value set of DP.

Data Property Instantiation: A short form *Data Property Definitions* (DPD) follows the pattern (u, A, `:adt`) where u can be the name of a

class or an object property, and A ∈ {.A, .^A} is the name of an principal or meta data property and :adt ∈ ADT denotes an *Atomic Data Type* (ADT). The *Value Set Ranges* (VSR) of .A and .^A are identical: VSR(.A) = VSR(.^A). Thus with v ∈ VS(.A) ⊆ VSR(.A) every *Data Property Instantiation* (DPI) must map to one of the following rules where the symbol ⊨ relates the property definition to its allowable property instantiation.

– (u, A, :adt) ⊨ (x, A, v) where (x, »iof, u) or x ∈ ∴u
– (◊OP, .A, :adt) ⊨ (»OPI, .A, v) for relators with (»OPI, »iof, ◊OP)

Value Assignment Propagation (VAP): In a hierarchy of universals ∴u a value v can be assigned to a data property starting from the class where it firstly has been declared in the form (u, a, :adt), for example, (^Car, .ManufDate, :Date) or (^Car, .^LaunchDate, :Date). If the name of a data property contains the prefix dot (.) it is a *Propagational Data Property* (PDP) and if it contains the prefix dot-circumflex (.^) it is a *Meta Data Property* (MDP). Value assignments of PDPs are propagated along the hierarchy of universals from the universal u, where the assignment was first made, to all subclasses that lie below u in the hierarchy ∴u and also to particulars that are connected to the universal u via »iof. According to Carvalho and Almeida [2], page 37 this is characterized by durability=∞. For MDPs there is no propagation of value assignments (durability=0), i.e., they can only by assigned once and are not inherited downwards. In addition, we provide the characteristic mutability for a data property for the case, that the value of an attribute needs to be forbidden to change, once it has been assigned, for example (.warmblooded, .mutable, F) or (.Gender, .mutable, T). This corresponds to setting mutability=0 in [2]. For MDPs .^a we have in general that they must not instantiate into particulars with the only exception of Adminstrative Meta Data Properties as described in the Evaluation section (s. 5): If p is a particular then there must not exist a (p, .^a, v) ∈ KG. Another desirable characteristic to be modeled for data properties is that of *disjointness*. If a data property a is declared to be 'disjoint' then in a hierarchy of universals it is not allowed that a subclass c1 of a class c2 has a different value for a: if ∃ (c1, a, v1) ∈ KG then there may not exist a (c2, a, v2) ∈ KG with v1 ≠ v2. As a consequence, the classes c1 and c2 have to be remodeled in a way that they become siblings with respect to another superclass which they have in common. Examples:

– Be ^SportCar a direct or indirect subclass of ^Vehicle. The data property definition (^Vehicle, .MaxNbrOfSeats, :Integer) and the assignment in ^SportsCar with (^PassengerVehicle, .MaxNbrOfSeats, 7) effect that every subclass of ^PassengerVehicle also has a maxium of 7 seats and that

every particular of `^PassengerVehicle` or any particular of a subclass of `^PassengerVehicle` has also 7 seats at maxium by default.
- Value assignments for disjoint PDPs must not be overwritten. If the assignment `(^SportsCar, .NbrOfSeats, 2)` and `(^Formula1_Car, .NbrOfSeats, 1)` are applied then the class `^Formula1_Car` cannot be a subclass of `^SportsCar`, because the set of single-seaters cannot be a subset of two-seaters. I.e., the two classes have to be aligned as sibling classes below a common superclass.
- Value assignments for non-disjoint PDPs may be overwritten. I.e, a particular instantiates all values of the PDP of all superclasses.
- Be `(^^Resource, .^produced, :Integer)` a definition of an MDP. Then the attribute `.^produced` can be assigned in any universal or any particular of a universal in the hierarchy below `^^Resource`. E.g. `(^Lamborghini_Miura, .^produced, 763)` models the fact that `763` cars of type `^Lamborghini_Miura` have been produced. Since `.^produced` is an MDP its value is **not** propagated to subclass of `^Lamborghini_Miura_P400` because this never makes sense, and would actually be an error source. This also implies that it is not mutable `(.^produced, .mutable, F)`.

Object Property Instantiation: An *Object Property Definition* (OPD) has the short form (^C, ◊OP, ^D), where ^C is the domain class and ^D is the range class. Then there are the following possible *Object Property Instances* (OPI): (1) `(>P1, »OP, >P2)`, where `>P1` must be a particular instance of a class from .˙.^C and `>P2` must be a particular instance of a class from .˙.^D (^C and ^D can be identical), i.e. `(>P1, »OP, >P2)`, is appended as particular object property instance under `(^C, ◊OP, ^D)`. (2) `(>P, »OP, ^C)`, or also `(^C, »OP, >P)`, which means that connections between the particulars layer and the universals layer are possible.In the vehicle ontology in Figure 1, we find the following Object Property Definition (OPD): `opd = (^Vehicle, ◊hasCombustionEngine, ^CombustionEngine)`. To find all instances of the OPD, the subclass hierarchies of `c = ^Vehicle` and `d = ^CombustionEngine` including the particulars of the subclass hierarchies must be examined for OPIs of the form `(x, »hasCombustionEngine, y)`, where it must not simultaneously hold that `x = c` and `y = d`, otherwise the OPD itself would also be included. The superclass hierarchies of s and o are ˙.˙s and ˙.˙o. The sets of particulars of ˙.˙s and ˙.˙o are SOP(˙.˙s) and SOP(˙.˙o). Be >P ∈ SOP(.˙.^D).

Object Property Instantiation Rules: The *Object Property Instantiation Rules* (OPIR) can consequently be derived as follows, where $\Diamond_{OPOf} = \Diamond_{OP}^{-1}$. For *Particular-Particular Relationships* (PPR) and for the *Class-Particular Relationships* (CPR) and their inverses, we get:

- PPR: (^C,◊OP,^D) ⊨ (>P1,»OP,>P2); (^D,◊OPOf,^C) ⊨ (>P2,»OPOf,>P1)
- CPR: (^C,◊OP,^D) ⊨ (^C,»OP,>P); (^D,◊OPOf,^C) ⊨ (>P,»OPOf,^C)

4 Application

In this chapter we show how our methodology can be applied in such a way as to reduce the complexity in ontology modeling, while fulfilling the requirements of Multi-Level Modeling (MLM).

Class-Particular Relations: The class ^Lamborghini_Miura in Figure 1 is a subclass of the class ^SportsCar, which itself is a subclass and instance of the class ^PassengerVehicle. The short Object Property Definition (OPD) for ◊isDesignerOf is (^natPerson, ◊isDesignerOf, ^Car). The relationship (>NPS-Marcello_Gandini, »isDesignerOf_LM, ^Lamborghini_Miura) indicates that >NPS-Marcello_Gandini is the designer of the car model ^Lamborghini_Miura rather than each particular ^Lamborghini_Miura. This fulfills one of the essential requirements ([12], page 2 and 3) of multi-level modeling. Generally speaking, relationships between particulars and classes are permitted. The general instantiation principle here is illustrated by the following examples:

- (>Yo-Yo_Ma, »ExpertOf, ^Violin); (^Violin, »Expert, >Yo-Yo_Ma)
- (>NPS-Albert_Einstein, »isCreatorOf, ^Special_Theory_of_Relativity)

Powertype Subsumption: As one of the obstacles to having simpler conceptual models, we have identified the fact, that without a further differentiation of data properties instantiation it is not possible to merge pairs of classes such as (Phone, Phone_Model), (Bird, Bird_Species), (^Car, ^Car_Model), etc., where the second class is the powertype of the first class, e.g., ^Car_Model = Powertype (^Car). The reason for having class pairs such as (^Car, ^Car_model) is that both classes in Figure 2 have different kind of properties. The class ^Car has data properties such as .LicensePlateNbr, .Seats etc. while ^Car_Model has properties such as .YearOfDesign, .produced (number of produced cars) etc. Particulars pertaining to both classes ^Car and ^Car_Model are instantiated on the Particulars Layer (PL).

With our methodology it is now possible to subsume powertype classes below their base classes. Compared to Figure 2, the OntoGraph in Figure 3 shows that the ^Car_Model class and the ^Car class have been merged into the ^Car class. At the same time the particular >Lamborghini_Miura has been transformed into the class ^Lamborghini_Miura and the connecting object property has been changed from »iof to »subModelOf where »subModelOf ⊂ »subClassOf. This transformation was easily achieved by changing the

prefix > into ^ for all triples (s, p, o) ∈ KG where s = >Lamborghini_Miura or o = >Lamborghini_Miura. Consequently, the instance >Lamborghini_Miura moves from the particulars layer to the universals layer. In addition, the two object property definition (^Car, ◊Car_Model, ^Car_Model) and the object property instance (>Toms_Lambo, »Car_Model, >Lamborghini_Miura_P400) for ◊Car_Model are no longer needed.

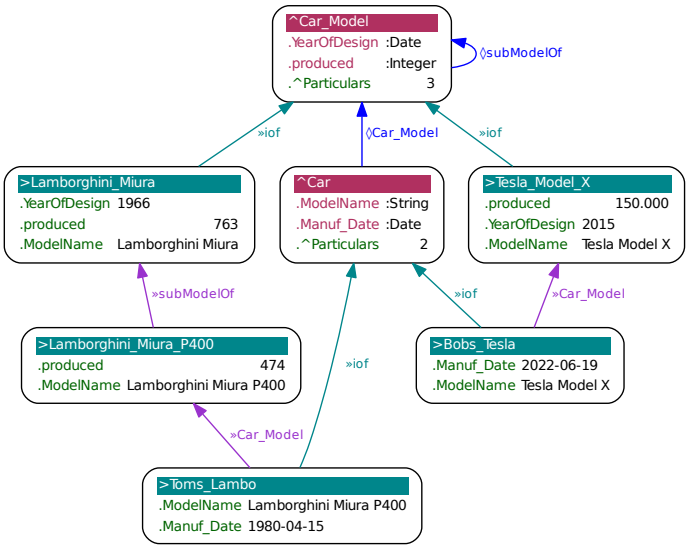

Fig. 2: Car Model with Materialization Pattern

The data property .Model_Name was modeled as a Propagation Data Property (PDP). Due to the Value Assignment Propagation (VAP), the triple (^Lamborghini_Miura, .Model_Name, Lamborghini Miura) it is propagated to all particulars of that car model, in this case to ^Lamborghini_Miura. In the subModel ^Lamborghini_Miura_P400 the value for .Model_Name is overwritten with Lamborghini Miura P400. Please note that this transformation from the model in Figure 3 to the model in Figure 2 can be reverted. Mixing the two class hierarchies results in the original powertype classes shifting downwards and the car's base classes moving upwards in the car hierarchy. Please also note that the data property YearOfDesign is modelled as an MDP in Figure 3 and as a PDP in Figure 1. The latter is an attribute of the object property instantiation »isDesignerOf_LM and therefore contains more information, namely who designed the car and not only when it was designed. The example in Figure 3 is also proof that the possibility exists of an extension with regard to non-uniform sub-hierarchies according to Neumayr et al. [12], page 2. This modeling of ^Car avoids fragmenta-

tion and redundancy since every piece of information concerning domain objects are now described locally to `^Car` objects.

Ultimately, as called for in [7] the modeler is released as far as possible from deciding when to model instances as classes or as particulars. In comparison, by taking our approach, there is not necessity to explicitly declare a subclass as a class. For example, the instance character of `^Tesla_Model_X` is revealed by assigning values to the data properties. The class character as the second facet arises because `^Tesla_Model_X` was modeled as `»subModelOf ^Car`.

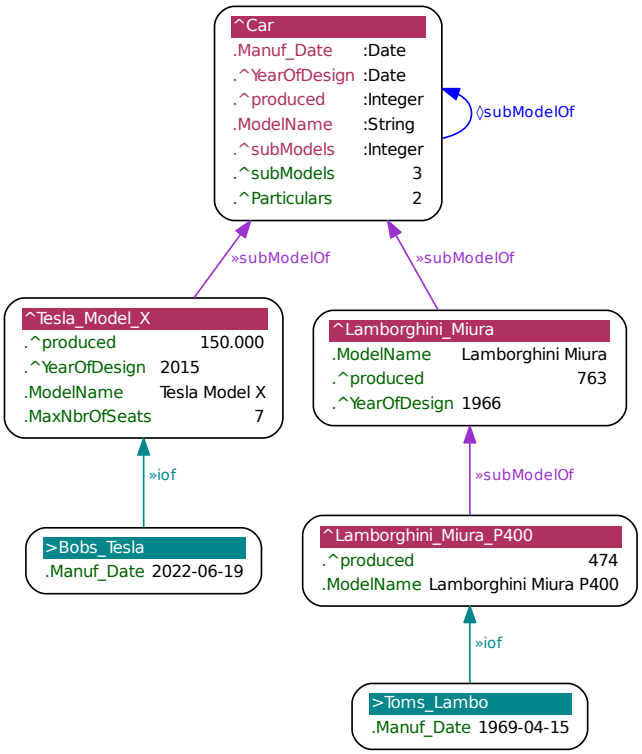

Fig. 3: Car Model with Powertype Subsumption

If `^Car_Model` is the *PowerType* (PT) of `^Car` then we can write `PT(^Car) = ^Car_Model`. The question then arises, is there also a `^Car_Model` powertype, for example, `^Car_Model_Model = PT(^Car_Model) = PT(PT(^Car))`? We think not, because what attributes can `PT(^Car_Model)` have? Of course, subsets could be formed from models of cars, but that could then be done using object properties such as ◊`CategoryOf`, ◊`SubordinateOf`, ◊`classifiedBy`, ◊`GroupOf`, ◊`OrderOf`, ◊`FamilyOf`, ◊`classifiedBy` or something similar at choice being subsets of ◊`subClassOf`. The example shows that, compared to other

instantiation methods such as deep or dual deep instantiation, we do not need to annotate additional properties of the data properties such as potency, mutability or durability. By using the method of Value Assignment Propagation (VAP) method for Propagation Data Properties (PDP) and by banning the application of VAP for Meta Data Properties (MDP) in all cases, we arrive at the targeted instantiation behavior.

What influence can the new methodologies have on the modeling style? First, it is noticeable that in conventional modeling there are fewer classes and more particulars. When subsuming the powertypes, the powertypes then become regular classes and become subsumed under their corresponding base types. It is also noticeable that the powertypes are universals by nature. So, the question is legitimate, why are they not modeled as classes from the outset. We have already argued that this is due to the lack of differentiation for the data properties found in traditional modeling. Since the VAP method now provides this, one modeling recommendation could be to model powertypes as classes and to subsume them under their base types from the start. This automatically creates the advantage of savings possibilities for object property instances.

5 Evaluation

We at first discuss the requirements for Multi-Level Modeling (MLM) that can be satisfied or which are no longer relevant. The reduction in the complexity of the modeling largely results from the applied naming conventions and the possibility of merging classes and saving object property instances by means of *Powertype Subsumption* (PS). Applying *Value Assignment Propagation* (VAP) shows that it is possible to manage with just two abstraction layers, namely M_0 = Particulars Layer (PL) and M_1 = Universals Layer (UL). The consequence of the instantiation method based on PS and VAP is that we can conclude that no *powertypes of types* or *types of types of types* exist.

Satisfying MLM Requirements: Many of the requirements posited for MLM in [8], [9] and [12] then turn out to be irrelevant or can be satisfied by our approach as follows: In the universals layer (UL) with an object property $\Diamond_{\texttt{ssr}} \in \texttt{SRR}$, it is possible to form arbitrary instantiation chains, since each relationship $(\texttt{u1}, \Diamond_{\texttt{ssr}}, \texttt{u2})$ between universals corresponds to an $\gg_{\texttt{iof}}$ relationship $(\texttt{u1}, \gg_{\texttt{iof}}, \texttt{u2})$. The principles of the organization of universals and particulars within the two layers UL and particulars layer (PL) are clearly defined by the property instantiation patterns. This also applies to the relationship between instances of the UL and PL. Fragmentation and redundancy are avoided by allowing base classes and their

powertype classes to be merged into the base class. Since the classification and assignment of objects to one of the levels is only controlled by means of naming conventions, the query methodology does not change. Therefore, the requirement stipulated in [12], page 3, i.e. the capacity to navigate between different levels of abstraction with queries, is fulfilled.

Parsimony: Compared to conventional modeling, the use of naming conventions and special instantiation methods in some cases actually saves a considerable number of triples. Taking the version containing the saved triples, a modeler can revert to traditional modeling at any time. The use of prefixes in the names of ontological concepts enables direct conclusions to be drawn about the nature of the concept, while facilitating parsimony through naming conventions. When defining property definitions, rather than have two triples, each applying the object properties `»hasDomain` and `»hasRange`, we only need one triple in each case. Using the powertype subsumption method, powertype classes can be subsumed under their base classes. As a result, at least two object property instances can be saved for each pair of connected particulars. Due to the instantiation mechanisms we introduced, it is only necessary to differentiate between the universal layer (UL) and the particular layer (PL). Unlike numerous other MLM approaches, this also removes most of the restrictions on how elements can be connected between layers. This reduces or even eliminates the risk of making layer mistakes during modeling. Compared to the modeling in [2], page 26, we regard our approach to be more flexible as it allows the merger of the base class `MobilePhone` and its powertype `MobilePhoneModel` into one class, while preserving the intended semantics at the same time. We solve the problem by defining `launchDate` as a Meta Data Property (MDP) of the class `MobilePhone`. Another argument for not distinguishing more than one level of abstraction for classes is provided by the definition of metaclasses: *In object-oriented programming, a metaclass is a class whose instances are classes.* As a result, every class would be a metaclass in any case and every class including `Class` would also be an instance of a class. In contrast to MOF, where `Class` appears on both levels 2 and 3, we only have the two layers M_0 = Particulars Layer (PL), and M_1 = Universals Layers (UL), while `Class` only appears on level M_1.

Singletons: Classes and particulars can also be connected by relators and thus connect instances between the universals layer and the particulars layers. The following example corresponds to modelling *Singleton Properties* (SP) as described in [14]. The relator `»isDesignerOf_LM` in Figure 1 is an instantiation of the object property ◊`isDesignerOf` and attributes the knowledge graph with the characteristics of a *Labeled Property Graph*:

— `(»isDesignerOf_LM, .YearOfDesign, 1966)` and
— `(>NPS-Marcello_Gandini, »isDesignerOf_LM, ^Lamborghini_Miura).`

Metadata: We distinguish betweeen domain-specific metadata (DMD) and administrative metadata (AMD). DMD form part of the application domain modeling, whereas AMD encompass information relating to the modelers, versions of the model, as well as technical and statistical information. Both types of meta data are modelled using Meta Data Properties (MDP). As a convention, we agree that AMD attributes are defined in `^^Resource`, while DMD attributes are defined in other universals. This is similar to the Dublin Core Metadata Initiative. In our approach, the mapping to the appropriate area can be easily altered by shifting the data property definitions (DPD) from `^^Resource` to a universal or vice versa. In the example `AnnotationAssertion( a:addedBy a:Dog "Seth MacFarlane" )` from OWL2 [9] the attribute `addedBy` can be regarded as a typical AMD attribute and we would place its definition in `^^Resource`.

Deep Instantiation: Compared to [3] we do not have to add and maintain a property such as *potency* to each data property. Our methodology dictates which instantiation rule is to be applied by using the ^ symbol in Meta Data Property Names or withholding it from Propagation Data Property Names. In our approach, as with deep instantiation, universals can exhibit both instance and type character simultaneously. In [4], page 9, the base classes `Book` and `Car` are modeled as subclasses of `ProductCategory`. We would merge the `ProductCategory` with the `Product` class and avoid a forced division into classification levels. The modelling in [5], page 2, could be simplified as follows: The powertype `BirdSpecies` can be subsumed under its base type `Bird` by converting the data properties in `BirdSpecies` from PDPs to MDPs and moving them into the base class `Bird`. `GoldenEagle` and `EmperorPinguin` become direct subclasses of `Bird`. So, instead of having two parallel class hierarchies for `Bird` and `BirdSpecies`, we have just one for `Bird`. This also resolves the deficiency in their modeling whereby the characteristics of all other species would have to be modeled redundantly as in `BirdSpecies`. We believe that our modeling methodology is in line with this and not only complements the authors' approach, but also extends it to streamline Multi-Level Modeling (MLM). Our approach makes it possible to solve the unresolved problem from [7], i.e. the inability to state that all `polar bears` have `white fur`. Using the VAP method we simply model `(^Polar_Bear, .ColorNameFur, white)`. We therefore do not require two different inheritance mechanisms for properties

[9] https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/#Metamodeling

such as `TypeInheritedProperty` and `BroaderInheritedProperty`, rather only the distinction of whether or not the value assignment is propagated.

6 Conclusion

Ontology engineering follows strict rules. We regard the triple representation of a knowledge graph as the form of representing knowledge which is common to at least all the approaches discussed in this paper. Triples can be transformed into the syntax of any modeling paradigm such as RDF/RDFS/OWL etc. The key outcome is that we have shown that, by applying a differentiated instantiation methodology, it is possible to remodel ontologies in a way that at the same time complexity can be reduced and transparency enhanced. The introduced naming conventions combined with instantiation rules are a minor necessary prerequisite to enable the advantages of our modeling approach to be utilized. To achieve the same goal, other approaches to multi-level modeling (MLM) require extra added information that cannot be stored in the same compact manner. Our approach recognizes classes as instance of other classes in general. Compared to other approaches our meta modeling approach only needs particulars and universals. For the modeler, the key decision regarding the right ontology design is narrowed to choosing whether a data property should be defined as a propagation data property (PDP) or as a meta data property (MDP).

Future Work: We will continue to work out how our presented instantiation methods can be used to systematize the partitioning of classes in such a way that considerable savings potential can be applied for storing ontologies. In addition, we will examine how the application of our methodology affects the simplification of ontology design patterns.

References

1. Humm B, Archer P, Bense H, Bernier C, Goetz C, Hoppe T, Schumann F, Siegel M, Wenning R, and Zender A. New Directions for Applied Knowledge-based AI and Machine Learning. Informatik Spektrum, Springer 2022. DOI: 10.1007/s00287-022-01513-9
2. Carvalho VA and Almeida JPA. Toward a Well-Founded Theory for Multi-Level Conceptual Modeling. 2016. Available from: https://nemo.inf.ufes.br/wp-content/papercite-data/pdf/toward_a_well_founded_theory_for_multi_level_conceptual_modeling_2016.pdf
3. Atkinson C and Kühne T. The Essence of Multilevel Metamodeling. International Conference on the Unified Modeling Language 2001 :19–33. Available from: https://link.springer.com/chapter/10.1007/3-540-45441-1_3

4. Neumayr B, Schrefl M, and Thalheim B. Modeling Techniques for Multi-Level Abstraction. 2008. DOI: 10.1007/978-3-642-17505-34. Available from: https://www.researchgate.net/publication/221024521_Modeling_Techniques_for_Multi-level_Abstraction
5. Guizzardi G, Almeida JPA, Guarino N, and Carvalho VA. Towards an Ontological Analysis of Powertypes. International Workshop on Formal Ontologies for Artificial Intelligence (FOFAI) 2015. Available from: https://www.researchgate.net/publication/279178175_Towards_an_Ontological_Analysis_of_Powertypes
6. Guizzardi G, Figueiredo G, and Poels MMHG. Ontology-Based Model Abstraction. IEEE Thirteen International Conference on Research Challenges in Information Science, Brussels 2019. DOI: 10.1109/RCIS.2019.8876971. Available from: https://www.researchgate.net/publication/332290797_Ontology-Based_Model_Abstraction
7. Bense H and Humm B. An Extensible Approach to Multi-Level Ontology Modelling. KMIS 2021, 13th International Conference on Knowledge Management and Information Systems 2021
8. Brasileiro F, Almeida JPA, Carvalho VA, and Guizzardi G. Applying a Multi-Level Modeling Theory to Assess Taxonomic Hierarchies in Wikidata. Wiki Workshop at 25th Int. Conference Companion World Wide Web 2016 :975–80
9. Frank U. Toward a New Paradigm of Conceptual Modeling and Information Systems Design. Business Information Systems Engineering 6(6) 2014 :319–37. Available from: https://www.researchgate.net/publication/275152525_Multilevel_Modeling_Toward_a_New_Paradigm_of_Conceptual_Modeling_and_Information_Systems_Design
10. Odell JJ. Power Types. Journal of Object-Oriented Programming, Volume 7(2) 1994 :8–12
11. Pirotte A, Zimanyi E, Assart D, and Yakusheva T. Materialization: a powerful and ubiquitous abstraction pattern. VLDB, Morgan Kaufmann 1994 :630–41
12. Neumayr B, Grün K, and Schrefl M. Multi-Level Domain Modeling with M-Objects and M-Relationships. 6th Asia-Pacific Conference on Conceptual Modeling 2009. Available from: https://www.researchgate.net/publication/220268481_Multi-Level_Domain_Modeling_with_M-Objects_and_M-Relationships
13. Bense H. The Unique Predication of Knowledge Elements and their Visualization and Factorization in Ontology Engineering. Kutz O, Garbacz P (eds.), Proceedings of the Eighth International Conference (FOIS 2014), Rio de Janeiro, Brazil, Sept. 22-25, 2014, DOI 10.3233/978-1-61499-438-1-251 2014 :241–50. DOI: 10.3233/978-1-61499-438-1-251. Available from: https://ebooks.iospress.nl/publication/37972
14. Nguyen V, Bodenreider O, Thirunarayan K, Gang Fu EB, Rosinach ŃuQ, Laura I. Furlong MD, and Sheth.Authors A. On Reasoning with RDF Statements about-Statements using Singleton Property Triples. 2015. Available from: https://arxiv.org/pdf/1509.04513.pdf