TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik - Institut für Systemarchitektur

TECHNISCHE BERICHTE
TECHNICAL REPORTS

ISSN 1430-211X

TUD-FI14-01 März 2014

Alexandr Savinov
Institut für Systemarchitektur, TU Dresden

Concept-Oriented Model and Nested Partially Ordered Sets

Technische Universität Dresden
Institut für Systemarchitektur
Professur für Datenbanken
Fakultät Informatik, 01062 Dresden, Germany
http://wwwdb.inf.tu-dresden.de/

Concept-Oriented Model and Nested Partially Ordered Sets

Alexandr Savinov

Technische Universität Dresden

http://conceptoriented.org/savinov
savinov@conceptoriented.org

Abstract. Concept-oriented model of data (COM) has been recently defined syntactically by means of the concept-oriented query language (COQL). In this paper we propose a formal embodiment of this model, called nested partially ordered sets (nested posets), and demonstrate how it is connected with its syntactic counterpart. Nested poset is a novel formal construct that can be viewed either as a nested set with partial order relation established on its elements or as a conventional poset where elements can themselves be posets. An element of a nested poset is defined as a couple consisting of one identity tuple and one entity tuple. We formally define main operations on nested posets and demonstrate their usefulness in solving typical data management and analysis tasks such as logic navigation, constraint propagation, inference and multidimensional analysis.

Keywords: data models; unified models; data analysis; query languages

1 Introduction

A model can be viewed as a mathematical description of a world aspect (Kaburlasos, 2006). For models of data, the primary concern is how to organize a set of elements by choosing an appropriate *structure* which allows us to break one large set into smaller groups so that the data can be represented, accessed and analyzed more productively. For a general purpose data model, the key problem is in finding the simplest and most natural structure and operations for implementing a wide variety of patterns of thought and mechanisms currently existing in data modeling.

Recently, a new data model has been proposed (Savinov, 2009, 2011b, 2012a, 2014b), called the concept-oriented model (COM), with the purpose to unify the existing data modeling methods and analysis techniques. It is based on three novel structural principles distinguishing it from other models: (i) *duality principle* assumes that an element is an identity-entity couple; (ii) *inclusion principle* postulates that all elements exist in a hierarchy; (iii) and *order principle* postulates that all elements are partially ordered. The main purpose of the duality principle is to answer the question *how* elements exist and how they are identified. The novelty is that this principle treats identities and entities as two symmetric constituents within one element. The main purpose of the inclusion principle and nested structure is to answer the question *where* elements exist. Each element is identified with respect to its parent element. Such hierarchical addresses consisting of several segments are analogous to conventional postal addresses or URLs. One novel feature is that inclusion relation treats inheritance (IS-A) as a particular case of containment (IS-IN) by eliminating the need in having two separate relations. The main purpose of partial order is to answer the question what is the *meaning* of an element. It provides a mechanism of semantic characterization where semantics of an element depends on its relative position among other elements in a partially ordered set. The main benefit of the inclusion principle is that one and the same formal structure can be treated from quite different points of view like multidimensional modeling and entity-relationship modeling.

A data model can be defined in two major ways: syntactically as a language and mathematically as some formal setting. Earlier, COM has been defined using its concept-oriented query language (COQL) (Savinov, 2006a, 2011a, 2012a, 2014a) which can be viewed as its syntactic embodiment. This language is based on a novel construct, called *concept* (hence the name of the model), which generalizes conventional classes and is used for modeling data types. Concepts participate simultaneously in two relations: *inclusion* and *partial order*. Inclusion generalizes inheritance while partial ordering of concepts is a new feature. The main benefit of COQL is that it allows for easily formulating complex analytical ad-hoc queries using a novel arrow notation without join and goup-by operators as well as without assigning such roles as dimension, measure, fact and cube.

The main purpose of this paper is to introduce a *formal* basis for COM. We propose a new setting, called *nested partially ordered sets* (nested posets), which is a formal embodiment of this model. The structure of a nested poset is illustrated in Fig. 1. Elements of nested posets are identity-entity couples which are drawn as pairs of two boxes where the rounded (grey) rectangle denotes identity and the normal (white) rectangle denotes entity. The nested structure of inclusion relation spreads horizontally which means that an element is included in its parent element which is position on the left of it. The partially ordered structure spreads vertically and is denoted by upward arrows leading from an element to its greater elements. Greater elements are positioned over this element and lesser elements are positioned under this element. This structure can be produced from a conventional nested set if we additionally assume that its elements are partially ordered. Alternatively, it can be produced from a conventional poset by assuming that its members can be (nested) posets themselves.

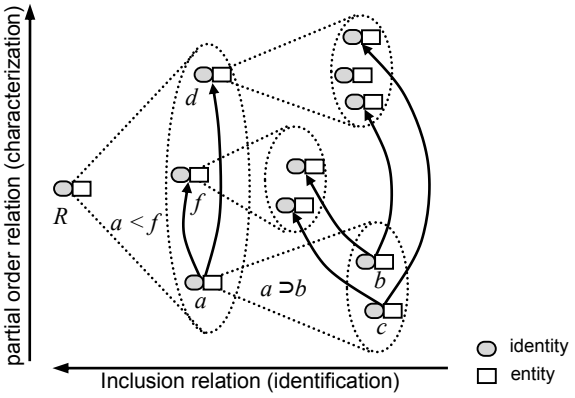

Figure 1 Structure of a nested partially ordered set.

2 Motivation and Goals

COM is an attempt to change the way data is viewed and manipulated by developing a simple and natural unified model which reduces a wide variety of existing data modeling methods and patterns to a few principles. COM is inspired in large part by deep incongruities between different views and approaches to data modeling and analysis (impedance mismatches) shortly outlined below in this section.

Identity vs. entity modeling. One basic belief is that it is entity that should be in the focus of data modeling. Therefore, almost all existing data models have a strong bias towards modeling entities while identities have been considered important yet secondary elements doomed to remain in their shadow. Therefore, identities are either provided by the platform in the form of primitive elements (Abiteboul & Kanellakis, 1998; Eliassen & Karlsen, 1991; Kent, 1991; Khoshafian & Copeland, 1986; Wieringa & de Jonge, 1995) or they are modeled by means of entity attributes (keys) assigned a special role. COM eliminates this asymmetry by making identities and entities equally important parts of an element *both* being in the focus of data modeling. Particularly, we can well create a model consisting of only identities (without entities) but in the general case identity modeling and entity modeling are supposed to be two equally important orthogonal concerns.

Value vs. object modeling. Values (transient data passed by-copy) and objects (persistent data passed by-reference) have always been considered two separate branches: either we model a value domain or we model object types. A typical example is the object-relational model (ORM) (Kim et al, 1990; Stonebraker et al, 1990) where user-defined types (sets of values or domains) are described separately from relations (sets of tuples). A fundamental question arises: if both domains and relations are sets then why to model and manipulate them using completely different constructs? In other words, if in mathematics there is only one kind of set then why do we need two different kinds of sets in traditional data modeling? COM resolves this problem and eliminates this separation by considering only one kind of sets which consist of value-object (identity-entity) *couples*. In COM, there is a single type hierarchy independent of whether a type describes values or objects. As a consequence, attributes are typed independent of whether they are of value type or object (records, rows) type. In relational terms, both relations and attribute domains are described using a single type hierarchy.

Instance-based vs. set-based modeling. In most models the notions of individual elements and sets of elements are principally separated. For example, tuples and objects are viewed as individual data elements which are not sets (of other tuples and objects). A relation, on the other hand, is a set which however is not treated and operated like a normal data element (tuple or object). Object data models (Abiteboul & Kanellakis, 1998; Dittrich, 1986) are traditionally more instance-oriented while the relational model (Codd, 1970) is set-oriented. The goal of COM is to eliminate these differences so that any instance is inherently a set (of other instances) and hence sets are also normal instances. Data management is then significantly simplified because it is reduced to maintaining only set membership while other interpretations (like having properties or relationships) are interpretations of this relation. Ideally, data manipulations should be reduced to only two operations: add an element to a set and remove an element from a set (where the set is another element). Another motivation is that any element should have a space (set) where it exists because a thing *in vacuo* outside of any space, domain, context or scope is regarded as nonsense. This allows us to bring structure into the space of elements which is very important when modeling huge (including web scale) databases where an element has a complex identity specifying its location in the global space.

Transactional vs. analytical modeling. There exist numerous approaches to modeling multidimensional data which are mainly driven by the demand from analytical applications and data warehouses (Pedersen, 2009). The problem is that the main assumptions and techniques behind analytical models are different from those behind transactional models (Conn, 2005). Most conventional database systems provide very limited analytical functions (Codd, 1993; Finkelstein, 1995; Kimball & Strehlo, 1994) while analytical systems are not intended for transactional processing. COM is aimed to integrate transactional and analytical views on data and to create a unified data model which could be used for both purposes. Creating such a model is driven by the demand from the industry where real-time analysis on transactional data is a very acute problem. Many currently existing solutions can provide hybrid *storage* for transactional and analytical operations but the problem is that data modeling and querying is still split into two parts.

The main notion in analytical models is that of dimension but if we ask the question how many dimensions a relational schema has then the answer most probably will be that it is not defined. The goal of COM in this context is to stop thinking of dimensions as something that is added separately at later stages of modeling and system development when it is necessary to solve some specific analytical task. We address this problem by rethinking dimensions and dimensionality (degrees of freedom) as first-class notions of the model which play a primary role for describing *both* transactional and analytical aspects. In COM, data is thought of as originally existing in multidimensional space (Savinov, 2005a) so that we always can say what coordinates this element has and how many dimensions this schema has. Also, existing multidimensional models provide weak support for ad-hoc agile analytics because they are intended for describing predefined (OLAP) analysis scenarios expressed in terms of cubes, dimensions, measures and aggregation types. The goal here is to develop an analytics-friendly model which is application-independent and is not restricted by the classical OLAP-type of analysis.

Conceptual vs. logical modeling. The main purpose of conceptual models is to provide richer mechanisms and constructs representing complex domain-specific concepts and relationships with the goal "to respond to queries and other transactions in a more intelligent manner" (Codd, 1979). Semantic relations represented at conceptual level as well as the corresponding queries need to be translated to the logical level. It is a highly ambiguous and non-trivial procedure due the existence of the semantic gap (Pardillo et al, 2008; Rizzi et al, 2006) between different levels of representation. In this context, the goal of COM is to make semantics integral part of the logical data model so that the database can be directly used for *reasoning* about data and maintaining its consistency. In particular, if we impose constraints in one part of the database then we want it to automatically infer the result in another part of the database without specifying how these constraints have to be propagated.

Data modeling vs. programming. There exists a well known and deeply rooted incongruity between programming and data modeling (Atkinson & Buneman, 1987). COM initially has been developed in parallel with a novel approach to programming, called concept-oriented programming (COP) (Savinov, 2014c, 2012c, 2008). In particular, concepts (generalized classes) and inclusion (generalized inheritance) are used in both COM and COP. Structurally, COM can be defined as COP plus partial order. This makes data modeling and programming much closer.

3 Model Structure

3.1 Duality

3.1.1 Tuples and Dimensions

In most existing data models, the minimum data unit is a value also called atomic, primitive, system or platform-specific value. More complex data elements are produced by means of tuples which are treated in their accepted mathematical sense by capturing the notion of an ordered list. They are denoted by either angle brackets ⟨…⟩ or parentheses (…) enclosing their members which are either primitive values or other tuples. A tuple consisting of *n* members is called *n*-tuple and *n* is called its arity.

Tuple members are distinguished by their position also called attribute, column, property, slot, variable or characteristic. In COM, this member position is referred to as a (simple) *dimension*. Dimension and the member are separated by colon. The tuple where a dimension is defined is called a *source* and the member represented by this dimension is called a *destination*. If $e = \langle \dots, x{:}\, a, \dots \rangle$ is a source tuple then *x* is a dimension and *a* is a member stored in this dimension. The source of the dimension *x* is denoted as $\mathrm{Src}(x) = e$ and its destination is denoted as $\mathrm{Dst}(x) = a$.

A *complex dimension* is a sequence of simple dimensions $x_1.\cdots.x_k$ where each next dimension starts where the previous dimension ends: $\mathrm{Src}(x_i) = \mathrm{Dst}(x_{i-1}), \quad i = 2,3,\dots,k$. The number k of constituents in a complex dimension is referred to as a *rank*.

3.1.2 Identities and Entities

The described representation by tuples has one general drawback from computer science point of view: it does not describe the mechanism of tuple membership, that is, we do not know what it means for a tuple to have another tuple as a member. If we assume that tuples are values (composed of other values) then we get tuple membership by-value as used in XML, JSON and other message formats. If we assume that tuples are included by-reference then any tuple is made up of references (which are primitive elements) similar to OOP and there is no possibility to build tuples containing complex values. Obviously, both mechanisms are needed in a good data model but the problem is that tuples do not allow us to distinguish and freely vary between them.

To overcome this fundamental limitation, COM introduces two kinds of tuples: *identity tuples* and *entity tuples*. Identity tuples are always *values* which are passed only by-copy. They do not have their own location, address, reference or any other kind of indirect representation. Rather, the purpose of identity tuples is to themselves serve as locations or references. There is no possibility to share an identity tuple – it can be only copied. Entity tuples are *objects* which can be passed only by-reference and there is no way to include an entity tuple in another tuple by-value. As a consequence, entity tuples are considered persistent elements which are *shared* among other tuples by storing the same reference. Identity tuples are transient data because they represent what can be transferred and hence they represent units of data exchange.

The principled separation of two kinds of tuples and treating them as symmetric elements of the model is an important but not sufficient condition for a good representation mechanism. The most important assumption made by COM is that these two kinds of tuples are two sides of *one* thing and identity tuples *are* actually references used to represent entity tuples. More specifically, the duality principle postulates that *the minimum data unit is a couple of one identity tuple and one entity tuple*. To distinguish these two constituents, identity tuples are written in angle brackets and entity tuple are written in parentheses. One identity-entity couple is referred to as a *segment*:

[segment] $s = \langle \dots \rangle(\dots)$ (1)

Given a segment *s*, function $\mathrm{id}(s)$ returns its identity tuple and function $\mathrm{en}(s)$ returns its entity tuple.

If segment *a* is a member in tuple *e*, $e = \langle \dots, a, \dots \rangle$, then only $\mathrm{id}(a)$ is included in *e* by-value while $\mathrm{en}(a)$ is represented and accessed by-reference. If all entity parts are empty then we can construct complex values composed of other values with empty persistent part. If all identity parts are primitive references then we get the second extreme case where all data are persistent objects. However, the

main advantage of this approach is that we can freely vary what data belongs to transient (non-shared) part and what data belongs to persistent (shared) part. Another advantage is that we can unite two branches: modeling value domains and modeling relations. COM considers only one type of domains where elements are identity-entity couples without the separation between value domains and relations as it is done in RM (Codd, 1970) and ORM (Kim et al, 1990; Stonebraker et al, 1990).

3.1.3 Concepts

To model segment types, COQL introduces a novel construct, called *concept*, which is defined as a couple of two classes: one *identity class* and one *entity class*. Fields of the identity and entity classes are referred to as *dimensions*. Concepts generalize conventional classes and are used precisely where classes are used for declaring types of elements. For example, bank accounts can be described by a concept which has account number as its identity dimension and current balance as its entity dimension:

```
CONCEPT Account
  IDENTITY
    CHAR(10) accNo
  ENTITY
    DOUBLE balance
```

Any variable or field of this concept will automatically store an account number and any access to this element will be performed using this identity. Note that identity is not a primary key (PK) because PK is composed of some attributes of the entity. Here is how a relation type could be defined using primary keys:

```
RELATION TYPE Account
  IDENTITY // Empty
  ENTITY
    PRIMARY KEY CHAR(10) accNo
    DOUBLE balance
```

Importantly, these are not syntactic differences. An instance of the `Account` concept has only one attribute in its entity (persistent part) and a relation tuple of type `Account` has two persistent attributes one of which is used as a key. COM identities can be viewed as user-defined surrogates (Codd, 1979; Hall et al, 1976) which are made part of an element type and which are also used as value types if the entity type is empty (Savinov, 2011b).

If entity class is empty then the identity class describes values because it is not intended to represent anything. For example, a currency amount could be defined as the following concept:

```
CONCEPT Amount
  IDENTITY
    DOUBLE amount
    CHAR(3) currency // "EUR", "USD" etc.
  ENTITY // No entity => identity is a value
```

Obviously, it is equivalent to user-defined types in ORM. The difference is that concepts are more general and can be used to define *both* identity (value) types and entity (relation) types in combination, that is, we can choose which attributes are transient values and which belong to the persistent part (object).

3.2 Inclusion

3.2.1 Extension and Nested Sets

In the previous section we answered the question *how* elements exist by assuming that an element is an identity-entity couple and its identity manifests the fact of existence. In this section, we answer the question *where* elements exist. COM assumes that elements are not able to exist in isolation outside of any external space, context or scope. Therefore, the inclusion principle postulates that *any element is included in some other element with respect to which it is identified*. Indeed, if identity is an address segment then it is reasonable to assume that any address segment is specified with respect to some parent address segment just because an address without a reference point is meaningless.

COM introduces a new operation, called *extension* denoted by colon: if a is an element and b is a segment then $e = a{:}\,b$ is extension of a with b. Element a is called a *base* element and new element e is called an *extended* element. Since the base element can also be an extension of its own base element, elements can be represented as a sequence of segments starting from some initial segment and ending with the last extension: $e = a{:}\,b{:}\cdots{:}\,c$ where $a, b, \ldots, c$ are segments. Just as with tuple membership, only identity parts are concatenated by-value and this sequence of identities is referred to as a *complex identity*. It is analogous to postal addresses with the main difference that any identity has an associated entity.

Extension is an operator which allows us to produce new elements by extending existing ones. Semantically, extensions are interpreted as more specific elements while the base element is considered a more general element. It is analogous to inheritance relation with the difference that we apply it to couples and elements exist in a hierarchy so that parent entities are shared among their children.

Extension can be also semantically interpreted in terms of set inclusion. The connection between extension operator ':' and strict inclusion relation $\subset$ is established in the set nesting principle:

[set nesting] $a{:}\,b \subset_1 a$, where $b \neq \langle\rangle$ (2)

Here $\subset_1$ denotes immediate inclusion (inclusion of rank 1) and if $e \subset_1 a$ then e (extension) is referred to as a *sub-element* and a (base) is referred to as a *super-element*. Thus any (non-empty) extension of an element is immediately included in it. In other words, to be a member in a set means to extend this set. Semantically, to be a member of a set means to be more specific than this set (more specific elements are subsets of more general elements).

This principle allows us to define all elements of the model as extensions of one root element and then induce a nested set $\langle R, \subset \rangle$. Conversely, if we take a nested set $\langle R, \subset \rangle$ then it can be represented as a number of elements each defined as an extension of its immediate super-element. Normally we choose one special element R, called the root or the universe of discourse, which does not extend any other element (this external context is ignored). Then all elements are uniquely represented as extensions of the root R. and all elements are subsets of this universe of discourse. Fig. 2 is an example of a nested set graphically represented using tree and Euler diagrams.

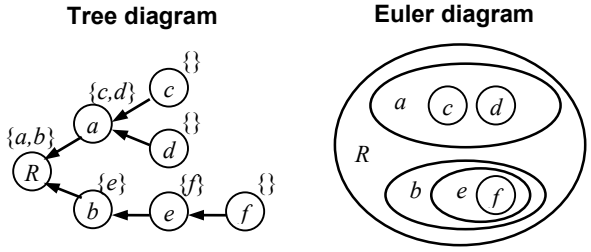

Figure 2 Two graphical representations of a nested set.

3.2.2 Concept Inclusion

Syntactically, inclusion is declared like inheritance by using the keyword 'IN'. For example, since savings accounts are always identified with respect to their main account my means of additional two digits, we use inclusion to describe this relation:

```
CONCEPT Savings IN Account
  IDENTITY
    CHAR(2) subAccNo
  ENTITY
    Amount savBalance
```

If we define a variable of this concept

```
Savings savAcc; // = "1234567890":"02"
```

then the value stored in it will consist of two segments: main account number and relative savings account number. Note that the `savBalance` dimension contains a value of the user-defined type `Amount` (with currency and balance). Here we see a benefit of duality and concepts: we can freely vary the structure of the `Amount` concept from values to objects or using their mix but all other concepts where it is used (like `Savings`) need not be updated.

The most important difference from classical inheritance is that elements in COM exist within a hierarchy rather than in a flat space. It is a direct consequence of introducing relative identities (like savings account number) which distinguish children only in the parent domain rather than in the global scope. In our example, one main account may have many savings accounts as well as many other possible types of sub-accounts. However, it is still possible to model classical inheritance if children do not define their identity. In this case only one child is possible (because they cannot be distinguished in the parent context) which is equivalent to classical inheritance. For example, an account with special privileges could be defined as follows:

```
CONCEPT Privileged IN Account
  IDENTITY // No identity => inheritance
  ENTITY
    Amount bonus // Very large bonuses
```

It is essentially an extension of the `Account` concept as it is treated in object-oriented approaches because we simply add an additional attribute to the entity without having a hierarchy of elements. The same can be done with value types when entity class is empty. For example, we could define a type of amount with date as follows:

```
CONCEPT DatedAmount IN Amount
  IDENTITY
    Date date
  ENTITY
```

This example is equivalent to the conventional extension of user-defined types but in COM it is only a particular case. In the general case, the advantage of inclusion is that it allows for modeling a hierarchy where any element is a set of its child elements and all elements have domain-specific addresses. Inclusion in COM assumes that IS-A is a particular case of IS-IN which significantly simplifies data modeling because fewer relations are used.

3.3 Partial Order

3.3.1 Tuple Ordering

New elements in COM (identity-entity couples) can be constructed either using extension operator ':' or tuples $\langle \dots \rangle$. Semantically, tuples in COM are interpreted as more specific elements with respect to the elements they combine. To guarantee consistency of this semantic interpretation we need to impose additional constraints on the structure of tuples. Analogously to extension which induces inclusion relation, the representation by tuples induces strict partial order relation '<' according to the tuple ordering principle:

[tuple ordering] $\langle \dots, e, \dots \rangle <_1 e$ (3)

Here $<_1$ means 'immediately less than' relation ('less than' of rank 1) and if $a < b$ then a is referred to as a lesser element and b is referred to as a greater element. According to this principle, tuple members are immediately greater than the tuple they are in. And conversely, a tuple is an immediately lesser element for all the elements it is composed of. (It is valid for both identity and entity parts.)

This principle allows us to define all elements as tuples satisfying the conditions of irreflexivity and transitivity, and then induce a partially ordered set $\langle R, < \rangle$. Conversely, if we take a partially ordered set $\langle R, < \rangle$ then it can be represented as a number of tuples combining their immediate greater elements. The greatest elements of this poset are primitive values all other elements are directly or indirectly composed of. The least elements are not used in any other tuple.

An important interpretation of partial order is via set inclusion which is referred to as inclusion by-reference to distinguish it from inclusion by-value induced by extension operator. It is assumed that lesser elements are included in their greater elements.

It is frequently convenient to consider a lattice of elements which (in finite case) has two special elements. The greatest element g, called *top*, is greater than any other element of the set: $\forall a \in R,\ a < g$. The least element l, called *bottom*, is less than any other element of the set: $\forall a \in R,\ l < a$. The number of immediate greater elements of this element is referred to as *arity* (also dimensionality, intention or valency). Fig. 3 is an example of a poset graphically represented using a Hasse diagram where an element is drawn under its immediate greater elements and is connected with them by edges.

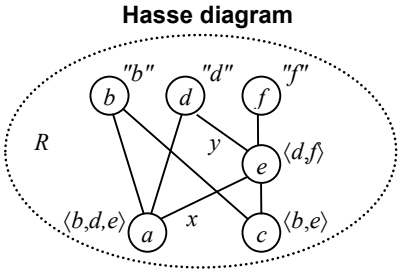

Figure 3 An example of a partially ordered set.

3.3.2 Concept Ordering

Let us assume that an employee is supposed to work at some department which means that the `Employee` concept has a dimension of concept `Department`:

```
CONCEPT Employee
    CHAR(64) name
    Department dept // Greater concept
```

This looks like a quite conventional definition but in COM it means that `Employee` is a lesser concept and `Department` is a greater concept.

3.4 Concept-Oriented Model

3.4.1 Inclusion as Order Constraint

Element participate in two relations simultaneously:

- inclusion relation $\subset$ where it has one super-element and a number of sub-elements, and
- partial order relation $<$ where it has a number of greater and lesser elements

Yet, semantically they have similar interpretations. Sub-elements just as lesser elements are treated as subsets and more specific elements. In other words, given an element we can produce more specific elements in two ways: either by extending it or by including it in a tuple. In order to avoid semantic contradictions where one element is more specific in one relation and more general in the other relation we impose the following constraint:

[inclusion as order] $\forall a, b \in R,\ \neg(a \subset b \wedge b < a)$ (4)

This principle does not permit elements to reference their sub-elements which is quite natural. If concept A is included in concept B, $A \subset B$, then it is prohibited to have a dimension of concept A in B. For example, we cannot reference a savings account from the main account because `SavingsAccount` is included in `Account`:

```
CONCEPT Account
    Savings savAcc // Prohibited
```

3.4.2 Type Constraint

A unique feature of COM is that an extension may well reuse a dimension already existing in its base element and then the question is what its possible values are. In other words, what values are possible for dimension x if its super-element returns value a for this dimension: $\langle \dots, x = a, \dots \rangle : \langle \dots, x = ?, \dots \rangle$. In order to restrict possible values of dimensions we impose the following constraint:

[type constraint] $e \subset b \;\Rightarrow\; e.x \subseteq b.x$ (5)

This principle means that any new extension can only append segments to the values returned by its parent. A value returned by a dimension always extends the value returned by the same dimension of the super-element. The deeper we go in the inclusion hierarchy, the more specific values are returned by dimensions. For example (Fig. 4), if element e is included in b, $e \subset b$, then the value $e.x$ must be included in (or equal to) the set represented by this same dimension of its superset, $b.x$. In particular, it is possible to have $e.x = d$ (as shown in the diagram) or $e.x = c$ or $e.x = a$ because both d and c are included in a.

At the level of concepts, type constraint means that dimension concept must be included in the concept of this same dimension declared by its super-concept. For example, assume that concept `C` is included in concept `B` which in turn is included in `A` and all three concepts have one common dimension `address`:

```
CONCEPT A
  ENTITY
    Country address

CONCEPT B IN A
  ENTITY
    City address

CONCEPT C IN B
  ENTITY
    Street address
```

Here type constraint means that concept `Street` must be included in concept `City` which in turn must be included in `Country`. If it is not so then this constraint is not satisfied. At the level of instances, this dimension will store only an extension part (one segment) with respect to the value stored in the parent. Concept `A` will store only country, concept `B` will store only city and concept `C` will stored only street of the whole address.

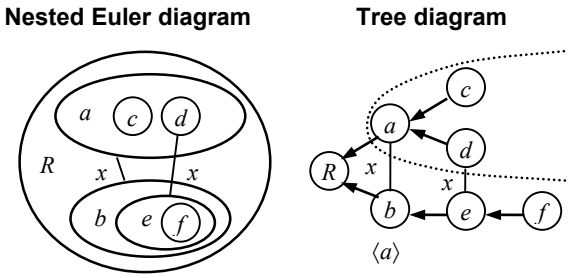

Figure 4 Type constraint.

3.4.3 Nested Partially Ordered Sets

We can now define a nested partially ordered set as a set with two binary relations and two constraints:

Definition 1 [Nested poset]. A *nested partially ordered set* $\langle R, \subset, < \rangle$ is a set R with strict inclusion $\subset$ and strict partial order $<$ relations on its elements which satisfy inclusion as order principle (3) and type constraint (4).

Nested posets are good for theoretical study but they hide some properties which are important in practical data modeling. In particular, this definition does not say how elements are represented and therefore the necessity of two relations is not obvious. For this reason, COM makes accent on representational issues by specifying how elements, inclusion and partial order are represented.

Definition 2 [Concept-oriented database]. A concept-oriented database is a nested poset represented according to the following principles:

- each data element is an identity-entity couple (1)
- strict inclusion $\subset$ is induced by extension operator (2)
- strict partial order $<$ is induced by tuples (3)

If we assume that elements are concepts then we get a concept-oriented schema. More specifically, a concept-oriented schema is a nested poset of concepts where all non-root concepts have one super-concept, and greater concepts are represented by dimension types.

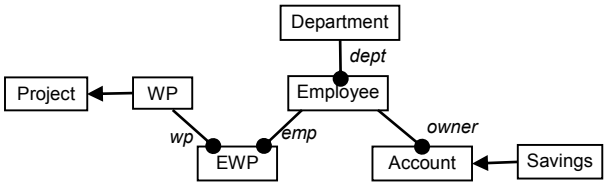

Figure 5 Example of a concept-oriented schema.

An example of a concept-oriented schema is shown in Fig. 5 (for simplicity, we do not show the separation of identities and entities). One employee belongs to one department (by-reference) and has many accounts (by-reference) each having many savings accounts (by-value). In addition, each employee works on several work packages (by-reference) each identified within some project. In terms of relationship multiplicity, upward and leftward lines mean many-to-one relationships while the opposite downward and rightward lines denote one-to-many relationships.

4 Operations

4.1 Projection and De-Projection

Querying data in COM is based on constraint propagation view which assumes that source constraints imposed in one part of the database schema can be propagated to another part of the schema. The main question is then how concretely constraints are propagated. A novel feature of COM is that constraints imposed on one set can be propagated either up (to greater sets) or down (to lesser sets). COM defines two operations, *projection* (denoted by right arrow) and *de-projection* (denoted by left arrow), which use dimensions for constraint propagation.

Definition 3 [Projection]. Given nested poset R and two its subsets $E \subseteq R$ and $D \subseteq R$, projection $E \to d \to D$ is defined as an operation that returns elements of D which are greater than elements from E along dimension d:

$$E \to d \to D = \{a \in D | \exists e \in E : e <_d a\}$$

Definition 4 [De-projection]. Given nested poset R and two its subsets $E \subseteq R$ and $F \subseteq R$, de-projection $E \leftarrow f \leftarrow F$ is defined as an operation that returns elements of F which are less than elements from E along dimension f:

$$E \leftarrow f \leftarrow F = \{b \in F | \exists e \in E : b <_f e\}$$

De-projection can be also defined as a set of elements which are projected to this set along the specified dimension. These definitions assume that the dimension can be complex. Note also that an element can be included in projection only one time even if it has many lesser elements. In terms of references, projection returns a set of elements referenced by the source elements and de-projection returns a set of elements which reference the source elements.

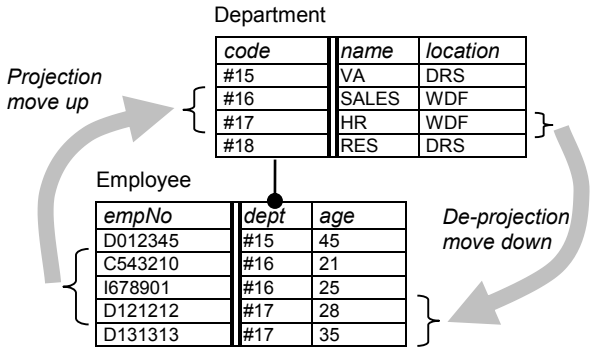

Figure 6 Projection and de-projection operations.

Projection and de-projection are used for set-based navigation (Savinov, 2005b) through the partially ordered set by moving up and down to greater or lesser elements. We can also project and de-project along inclusion hierarchy (horizontally) using 'super' as a special dimension name:

$E \rightarrow \text{super} \rightarrow D = \{a \in D | \exists e \in E : e \subset_1 a\}$

$E \leftarrow \text{super} \leftarrow F = \{b \in F | \exists e \in E : b \subset_1 e\}$

These operations allow us to get all immediate supersets or all immediate subsets (extensions) for the selected elements. A sequence of projection and de-projection operations is referred to as an *access path*. Access paths have a zig-zag form and provide the main means for propagating constraints from the source part of the model to the target part using its partially ordered structre. The approach based on projection and de-projection is referred to as arrow notation. It can be viewed as a set-based analogue of the conventional dot notation.

In COQL, a set of elements is written in parentheses by specifying its concept (or collection) name with constraints separated by bar symbol. For example, `(Employee|age<30)` is a set of all young employees. Projection operation is denoted by right arrow `'->'` followed by some its dimension name which is followed by a target set. For example, all departments for a set of young employees can be obtained as follows (Fig. 6):

```
(Employee | age<30) -> dept -> (Department)
```

De-projection is denoted by left arrow and is applied to the source set followed by a dimension of the target set. For example, given a department we can get all its employees:

```
(Department | code == "HR") <- dept <- (Employee)
```

To demonstrate how simple and natural this approach is let us consider a more complex request (Fig. 7) where we need to get all employees of the "DMA" department working on the "COM" project but assigned to less than 4 different work packages:

```
(Project | code == "COM") <- super <- (WP)                    ①
  <- wp <- (EWP)                                             ②
  -> emp -> (Employee | dept.code == "DMA" AND               ③
     this <- emp <- (EWP) -> wp -> (WP) < 4 )
```

This query consists of three major steps leading from `Project` to `EWP` and then to `Employee`. However, `Employee`s are selected using an internal access path for computing the number of assigned work packages:

```
this <- emp <- (EWP) -> wp -> (WP) < 4
```

(Comparing a set with a number is a shortcut for the COUNT aggregation function, that is, we compare the power of this set with this number.) Note that no joins and no group-bys are used here which are the main source of errors in SQL in many other join-based queries.

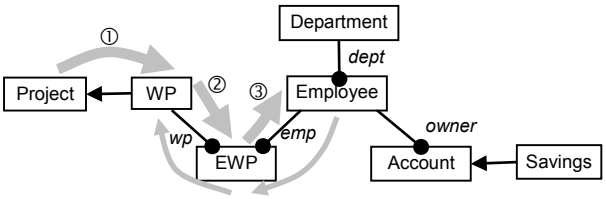

Figure 7 Example of access path.

4.2 Constraint Propagation and Inference

Above we have described a mechanism for constraint propagation along an *explicitly* specified path using projection and de-projection operations. In this section we describe a procedure, called inference, which can *automatically* propagate source constraints to the target. A simple example of such type of queries is where we would like to retrieve all employees for a project with no information on how it should be done which can be written as follows:

```
GIVEN (Project | code == "COM") // Constraint
GET (Employee) // Result
```

Answering such queries especially in the case of complex schemas is a highly non-trivial task because the database has to be "smart enough" to understand *how* to propagate the constraints through the model. Approaches relying on formal logic like semantic databases, description logics, and deductive databases rely on inference rules. In this case data and knowledge exist separately and are treated differently. An alternative approach consists in employing the structure of data for inference like it is done in the universal-relation model (URM) (Beeri et al, 1978). COM uses a novel approach which also belongs to this category and uses data itself as dependencies for carrying out inference. Its distinguishing feature is that it relies on partially ordered structure for that purpose. The basic idea is that lesser (more specific) data elements are treated as relationships or dependencies between their greater (more general) elements.

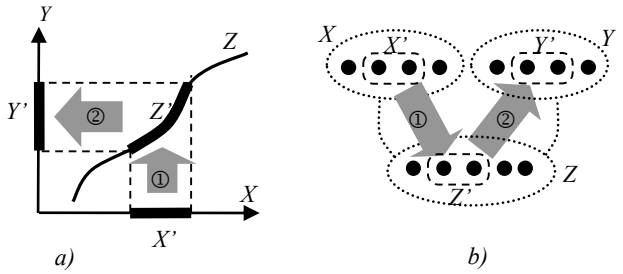

Figure 8 Inference in two-dimensional space.

Let us describe this procedure for the case of two domains where constraints on one domain have to be propagated to the other domain. If there is a functional dependency $y = f(x)$, then by choosing some $x \in X$ we can easily *infer* the corresponding $y \in Y$. If we now choose a subset $X' \subseteq X$ then the subset $Y' \subseteq Y$ can be inferred in two steps (Fig. 8a): 1) select $Z' \subseteq Z$ where $Z' = \{\langle x, y\rangle | x \in X', y = f(x)\} \subseteq Z = X \times Y$ is a subset of points on the plane Z, and 2) select $Y' \subseteq Y$ where $Y' = \{y | \langle x, y\rangle \in Z'\}$.

The crucial observation here is that the sets X, Y and Z are partially ordered according to the tuple ordering principle (3) because $z = \langle x, y\rangle$. This means that X and Y are greater elements, and Z is a lesser element: $Z <_x X$ and $Z <_y Y$ (see Fig. 8b). Therefore, the first step of the procedure is actually de-projection of the source set X' to the set Z by producing set Z', and the second step is projection of Z' to Y by producing Y'. If we now choose any dimension path (denoted by star symbol) then the inference procedure used in COM is defined as follows:

1. [De-projection] Source constraints X' are propagated down to the set Z using de-projection: $Z' = X' \leftarrow * \leftarrow Z$

2. [Projection] Constrained set Z' is propagated up to the target set Y using projection: $Z' = Z' \rightarrow * \rightarrow Y$.

In the case of n independent source constraints $X_1', X_2', \ldots, X_n'$ imposed on sets $X_1, X_2, \ldots, X_n$ the de-projection is computed as their intersection: $Z' = \bigcap X_i' \leftarrow * \leftarrow Z$.

In COQL, inference operator is denoted as `'<-*->'` which is a composition of de-projection and projection connected by arbitrary dimension path (star symbol). Using this operator, the last query from the previous section can be rewritten by applying two inference operators:

```
(Project | code == "COM")
  <-*-> (Employee | dept.code == "DMA" AND
  this <-*-> (WP) < 4
)
```

Here the database finds related employees by propagating the project element down to `EWP` (lesser collection) and then up to `Employee`. In addition, this operator is used to find a set of work packages for the current employee using (again) the `EWP` collection as a dependency.

This general inference procedure has several parameters. First, we can choose what dependency Z to use for inference. Normally, it is a bottom set in the database containing the most specific elements which directly or indirectly connect all other elements. However, complex database schemas may have many bottom sets and then we can either use all of them for inference or choose the most appropriate one. Second, we can vary de-projection and projection propagation paths. By default, constraints are propagated along all dimension paths but in some cases we might want to choose the most appropriate dimensions. Third, we can impose constraints representing additional (background) knowledge in the query (Savinov, 2006b, 2012b).

4.3 Product

Projection and de-projection take existing elements as input and return existing elements as output so that no new elements are produced. Yet, sometimes it is necessary to build new elements from existing ones. One such operation is the well known Cartesian product which returns new tuples built from the elements of the source sets $E_1, \ldots, E_n$:

$$E_1 \times \ldots \times E_n = \{e = \langle e_1, \ldots, e_n \rangle | e_i \in E_i, i = 1, \ldots, n\}$$

In COM, this classical operation has new properties. First, the new set (remember that sets are normal elements in COM) as well as all its members are lesser elements with respect to the source sets and their elements:

$$E_1 \times \ldots \times E_n < E_i,\ \langle e_1, \ldots, e_n \rangle < e_i \in E_i,\ i = 1, \ldots, n$$

This property is a consequence of the tuple ordering principle (3). Thus product in COM does not produce a completely new isolated set of elements. Rather, new elements become part of the model because they are defined in terms of this model (as new lesser elements).

Second, product consists of identity-entity couples and therefore tuple members belong either to identity or entity parts:

$$E_1 \times \ldots \times E_n = \{e = \langle e_1, \ldots, e_k \rangle (e_{k+1}, \ldots, e_n) | e_i \in E_i, i = 1, \ldots, n\}$$

Formally, this distribution between identity and entity parts has to be specified either explicitly or implicitly.

Third, arity of the product is equal to the number of source sets n. because the elements of the source sets are represented by their identities. For comparison, arity in the relational algebra is equal to the *sum* of the arities of the source sets because all the source attributes are included in the result tuple.

In COQL, product operation is implemented via `FOREACH` (also denoted by `CUBE` in some papers) statement followed by a sequence of source sets. For example, the following query returns all combinations of projects and departments:

```
FOREACH (Project, Department)
```

The structure of the return set is specified in the RETURN and additional constraints can be imposed in the WHERE statements:

```
FOREACH (Project p, Department d)
WHERE d.address.country == "DE"
RETURN p.name, p.type, d.name
```

This query selects only departments from Germany and returns three dimensions (instead of two identities p and d by default).

This type of query significantly simplifies analytical tasks like OLAP where it is necessary to compute some measure for a number of elements (cells of the cube). The measure is computed in the query body. For example, assume that EWP collection has a dimension pm which stores the number of person months an employee has in a work package. If we want to show total person months for each project and department then it can be done as follows:

```
FOREACH (Project p, Department d)
WHERE d.address.country == "DE"
BODY {
  Collection cell = p <-* (EWP) AND d <-* (EWP)
  DOUBLE measure = SUM( cell.pm )
}
RETURN p.name, d.name, measure
```

The body block is executed for each combination of project and department (instance variables p and d contain their identities). The first line finds all EWP elements belonging to the current project and department. (We use a shortcut that AND denotes intersection when applied to sets). We use star de-projection for constraint propagation so that an exact dimension path is not needed. The second line of the body finds total person months for the selected elements in the cell collection. Equivalently, the body of this query could be written as follows:

```
DOUBLE measure = SUM(
  (EWP | wp.super == p AND emp.dept == d).pm
)
```

First, we select all elements of EWP which belong to the current project and department.

```
(EWP | wp.super == p AND emp.dept == d)
```

(Here AND is used as usual as a logical conjunctive.) Then one dimension is selected and finally the SUM aggregation function computes the total number of person months for this cell.

5 Discussion

1) Why duality? The main benefit of the duality principle is that it unites value/identity modeling and object/entity modeling. Identities and entities are considered parts of one whole being modeled using one construct. Very informally, elements in COM are analogous to complex numbers in mathematics which also have two constituents but are manipulated as one whole. One consequence is that we can freely vary the "degree of sharing" by placing dimensions to be passed by-value or by-reference in identity class or entity class, respectively. As two extreme cases, we can describe a completely transient model where all elements are passed by-value (for example, message formats) and a completely persistent model where all elements are represented using the same type of platform-specific identity. This principle resolves the fundamental difficulty of traditional models with the necessity to have two kinds of sets: sets of values (domains) and sets of objects or tuples (relations). By uniting identities and entities within one thing, COM eliminates this separation and by using only one kind of sets. Another consequence of this principle is that identity modeling is not only made symmetric to but in many aspects more important than entity modeling. In COM, we first define how things in the problem domain are identified and only after that add persistent properties in entity part of the elements.

2) Why inclusion? The original design goal was to introduce hierarchical addresses for elements based on extension operator. The rationale behind our approach is that any address (identity) must have some parent address with respect to which it is defined and there can be many child addresses which extend this address. As a result, any element in the model gets its own unique hierarchical address which can be passed to other elements and then used to access its entity part. In this sense, COM can be viewed as a reincarnation of the hierarchical model on a new cycle of development. It is also similar to the hierarchical structure in XML documents where children are also included by-value (yet, without unique relative identities). By introducing address hierarchies we get two separate mechanisms with different design goals: containment and inheritance. Classical inheritance does not use the notion of addressing and instances are supposed to exist in a global flat space. On the other hand, containment does not employ the conception of reuse. Having two separate relations would make the model more complicated and the challenge was to express them within some more general mechanism. Here the most important contribution of COM is that it unites containment (with hierarchical address system) and inheritance. In COM, to be included in some container and to have an address within some scope means to extend it and to inherit its properties (a member of a set is an extension of this set). In terms of conceptual modeling, IS-A relation in COM is a particular case of IS-IN relation. Such a generalization has even more important consequences in COP (Savinov, 2014c, 2012c, 2008) where it underlies the mechanisms of dual methods (each method is defined twice in identity class and entity class) and reverse overriding strategy (parent methods can override child methods).

3) Why partial order? The short answer is that partial order is needed for data semantics where formal structure gets some meaning and all elements have formal definitions in terms of other elements. In particular, partial order is used for representing hierarchical multidimensional spaces (Savinov, 2005a). The meaning of an element is defined by its coordinates which in turn are treated as points with their own coordinates. It is therefore quite natural to prohibit cycles because points should not take themselves as coordinates. Cycles also produce an infinite number of dimensions (paths from bottom to top). Another interpretation of partial order is in terms of a generalization-specialization hierarchy and cycles in this case should also be avoided (for the same reason they are not allowed in inheritance hierarchies). Since the meaning of an element is defined by its greater elements, cycles will result in recursive definitions where a term is defined via itself. Also, if a greater element is interpreted as a set consisting of its lesser elements then we would like to avoid recursive membership where an element is (indirectly) its own member.

4) Modeling cycles: In many applications strict partial order is a too strong constraint especially in the case a element should reference itself. For example, if employees have a manager who is also an employee then we get a loop which does not produce any problem in practice. Therefore, self-references could be permitted in COM and it does not entail any difficulties when treated appropriately. Cycles also could be modeled if we introduce the mechanism of *inverse dimensions*, that is, dimensions which represent a lesser element (rather than a greater element by default). For example, if employees reference their department and the department references its manager then we get a cycle. To resolve this cycle we have to indicate which of these two concepts is more general by marking one dimension as an inverse dimension:

```
CONCEPT Department // Greater concept
  ENTITY
    inverse Employee manager // Inverse dim

CONCEPT Employee // Lesser concept
  ENTITY
    Department dept // Direct (normal) dim
    Employee assistant // Loop (self-reference)
```

Here `Department` is still greater than `Employee` because dimension `manager` is marked as pointing down. This information is used for automatic constraint propagation and inference where the system has to choose correct direction for constraint propagation (up or down). In the case of precise access paths where all dimensions are specified we can use arbitrary structure with cycles.

5) Why two relations? Semantically, both inclusion and partial order have similar meaning and can be used to express typical relationships like containment. Yet, this separation is very important and cannot be removed because we need some representation mechanism for elements. The main

difference between inclusion and partial order is that inclusion is implemented by-value using extension and partial order is implemented by-reference using tuples. If an element is included in its parent then it exists there permanently because its identity is defined with respect to the parent identity. If an element has a greater element then it also can be thought of as included in it. However, this inclusion is temporary and can be changed by changing the reference which points to the greater element. If we think of an element as existing in some space then there are two possibilities to define it: to include the element in the parent space by-value (as a whole) or to include it in several parent spaces by-reference using its dimensions. For example, earlier we assumed that an employee references the department where it works, which means that an employee is a member in the space of departments:

```
CONCEPT Employee
    CHAR(64) name
    Department dept // By-reference
```

The same semantics could be expressed using inclusion relation if employees are supposed to be *identified* with respect to the parent.

```
CONCEPT Employee IN Department // By-value
    CHAR(64) name
```

Here employees cannot (easily) change their department because it is part of their identity. Which alternative to choose is a matter of good design and is not defined by the model itself.

6) Why projection and de-projection: Most models rely on either join or dot notation for data connectivity. Formally, nothing prevents COM from using joins. However, joins have several negative properties (Savinov, 2012d):

- Join is a low level operation describing *how* to get a result rather than what result has to be produced.
- Join has several major purposes which cannot be distinguished from the syntax. For example, the join condition `WHERE A.id==B.id` says almost nothing about the purpose of the query
- Join is a cross-cutting concern which exposes the internal mechanics of relationships at the level of business logic. Writing and maintaining join-based queries is difficult and error-prone because any small change in the model structure entails numerous changes all over the queries.

Conventional references and dot notation are free of these drawbacks but they have other problems. First, primitive references are not part of the model and there is no way to define domain-specific formats. Second, dot notation is more instance-oriented and does not inherently support set operations. Therefore, dot notation is used mainly to follow a sequence of references while joins are used for more complex manipulations with data. COM proposes a novel approach to querying based on arrow notation with the following advantages:

- Projection and de-projection are set-based operations (in contrast to dot notation)
- Projection and de-projection are navigational operations (in contrast to join and similar to dot notation)
- Projection and de-projection hide their implementation and queries are more stable with respect to model alterations. They describe what has to be retrieved rather than how it has to be done
- Projection and de-projection are semantic operations because they are treated as movements: in containment hierarchy, between levels of detail, in general-specific hierarchy

6 Related Work

1) Identity and value modeling: Defining an element as an identity-entity couple is a novel feature which is absent in other models. Most data and programming models are entity-oriented by providing only primitive types for references. In the object-relational model (ORM) (Kim et al, 1990; Stonebraker et al, 1990), both values and relations (entities) are considered important for data

modeling but they are modeled separately: value domains are described using object-oriented means while entities are described using relational methods. In this sense, COM can be viewed as a generalization of ORM because this separation is removed so that both values and relation tuples are modeled using one construct. Primary keys are similar to COM identities because they both describe domain-specific identifiers. The difference is that primary keys are identifying entity properties (Eliassen & Karlsen, 1991; Wieringa & de Jonge, 1995) while COM identities are values. OIDs in object data models (Abiteboul & Kanellakis, 1998; Dittrich, 1986) are values but they have a platform-specific format. Yet, the distinguishing feature of COM is not only the possibility to define an arbitrary format of identities but making it together with the corresponding entity using one construct.

2) Modeling hierarchies: Inclusion principle makes COM similar to the hierarchical model (HDM) because in both cases elements exist in a hierarchy. The difference is that COM hierarchy is defined in terms of domain-specific addresses while HDM defines a hierarchy in platform-specific terms. Another important difference is that COM hierarchy generalizes inheritance relation and this feature makes it similar to object data models (ODM). However, ODMs use classical treatment of inheritance which entails the following asymmetry: classes exist in hierarchy while their instances exist in flat space. COM eliminates this asymmetry so that both concepts and their instances exist in hierarchy. It is also how inheritance is treated in prototype-based languages where "parents are shared parts of children" (Lieberman, 1986). However, COM implements instance hierarchy using identities and extension operator while in prototype-based languages the hierarchy is maintained by the compiler (each object segment has its own platform-specific reference). What is really new in COM is that it unites two views which earlier have existed separately: hierarchy as containment (IS-IN relation) and hierarchy as inheritance (IS-A relation). Inclusion makes it possible to introduce relation hierarchies in RM (if entities describe relation types). Moreover, modeling relation types and value types is unified because one inclusion relation is used simultaneously for both purposes which can be viewed as a generalization of ORM.

3) Partial order: Partial order has been widely used in conceptual modeling but not as a main principle but rather as a desirable property. In contrast, COM establishes partial order as one of its major principles and then derives other properties as consequences which is a novel approach. The only known work where partial order relation is laid at the foundation of data modeling is (Raymond, 1996) where "partial order database is simply a partial order". This approach focuses on manipulating many different partial orders and relies more on formal logic while COM focuses on manipulating elements within one nested poset with strong focus on dimensional modeling and analytical operations. Another difference is that it proceeds from the assumption that partial order underlies type hierarchies while COM uses inclusion relation for that purpose. COM relies on the principle that elements with an absent property are more general and elements with additional properties are more specific which is also studied in (Buneman et al, 1991; Zaniolo, 1984). However, this work is positioned as an extension of the relational model rather than an independent principle leading to a new model.

4) Multidimensional modeling and analysis: COM is very similar to multidimensional models (Pedersen, 2009) because both approaches view data as having many levels of detail and many dimensions. The difference is that COM is a general purpose model which can be used for both transactional and analytical applications. In contrast, standard OLAP models are designed for concrete type of analysis over some translational model. In COM, such OLAP roles as dimension, measure, facts and cube are not specified in the model because they are assumed to belong to the application level.

5) Conceptual modeling and semantics: Classical semantic and conceptual models (Hull & King, 1987) provide richer mechanisms and constructs for representing complex application-specific concepts and relationships. COM integrates logical and conceptual modeling by providing semantic features as its integral part. The main distinguishing feature is that semantics is represented by partially ordering all elements. The meaning of an element is defined by its greater elements which in turn have their greater elements so that all definitions are reduced to some basic terms (primitive values). In this case semantic relations (like containment, aggregation, generalization and classification) are not introduced as independent modeling constructs. Instead, one and the same structure (inclusion and partial order) is assigned different semantic interpretations. In terms of (Codd,

1979), a reference in COM represents an elementary semantic unit, "atomic semantics", while extension and tuple serve as a means for creating more complex semantic units, "molecular semantics". Another specific feature is that COM has *two* orthogonal relations: one is implemented by-value by extending identities and the other is implemented by-reference using tuples. Two relations are also used in (Smith & Smith, 1977) where an element aggregates other elements but at the same time it has a more general element. The difference is that COM uses inclusion and partial order relations as orthogonal structures instead of aggregation and generalization. COM is also characterized as an attributed-oriented and relationship-free model as opposed to fact-oriented models. Instead of explicitly introducing relationships, COM assumes that they are represented by lesser concepts (Savinov, 2012a).

6) Navigation and reasoning: Navigational features of COM are similar to those provided in the functional model (FDM) (Sibley & Kerschberg, 1977). This similarity is even stronger if we assume that partial order is represented by functions (rather than by tuples). De-projection operator in COM is analogous to the operator of inversing which is present in many models including FDM (inverse functions). Both COM and Description Logics (DL) rely on binary relations for data access and reasoning. However, DL uses formal logic which provides much more powerful reasoning mechanisms while COM provides more facilities for structuring data and then using these structures (rather than inference rules) for data access and inference. A novel feature of COM is that its inference mechanism relies on multidimensional treatment of data which is very useful in business intelligence. COM is similar to the universal-relation model (URM) (Beeri et al, 1978) because both rely on the structure of data for inference and query answering. URM could be obtained from COM by assuming that there are only two levels in the partially ordered structure (relations are immediate lesser elements for their attribute domains). In this case constraints imposed on attributes are propagated through the relations to other attributes. Bottom concept in COM would correspond to the universal relation in URM. In this sense, COM can be viewed as a new interpretation of the idea of having a universal relation.

7 Conclusion

Nested partially ordered sets introduced in this paper serve as a formal basis for the concept-oriented model. An element of a nested poset is defined as a couple of identity tuple and entity tuples by producing a nice yin-yang style of balance between two sides of one reality — entity modeling and identity modeling. Extension operator on elements induces inclusion relation between sets of elements which is used for modeling nested sets with a hierarchical address space. Tuples induce partial order relation among element which is then used to define projection and de-projection operations. The main advantage of the introduced formal setting is that it allows us to rigorously describe structural and operational properties of the model by generalizing many existing data modeling patterns and techniques.

8 References

Abiteboul, S. & Kanellakis, P.C. (1998). Object identity as a query language primitive. *Journal of the ACM (JACM)*, **45**(5), 798-842.

Atkinson, M. & Buneman, P. (1987). Types and persistence in database programming languages. *ACM Computing Surveys (CSUR)*, **19**(2), 105-70.

Beeri, C., Bernstein, P.A. & Goodman, N. (1978). A sophisticate's introduction to database normalization theory. In *Proc. VLDB'78* (pp. 113-124).

Buneman, P., Jung, A. & Ohori, A. (1991). Using powerdomains to generalize relational databases. *Theoretical Computer Science*, **91**(1), 23-56.

Codd, E. (1970). A relational model for large shared data banks. *Communications of the ACM*, **13**(6), 377-387.

Codd, E.F. (1979). Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, **4**(4), 397-434.

Codd, E.F. (1993). Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Technical report, E.F. Codd and Associates.

Conn, S.S. (2005). OLTP and OLAP data integration: A review of feasible implementation methods and architectures for real time data analysis. In *Proc. SoutheastCon* (pp. 515-520).

Dittrich, K.R. (1986). Object-oriented database systems: the notions and the issues. In *Proc. Intl. Workshop on Object-Oriented Database Systems* (pp. 2-4).

Eliassen, F. & Karlsen, R. (1991). Interoperability and object identity. *ACM SIGMOD Record*, **20**(4), 25-29.

Finkelstein, R. (1995, April). MDD: Database reaches the next dimension. *Database Programming and Design*, 27-38.

Hall, P., Owlett, J. & Todd, S. (1976). Relations and entities, In G.M. Nijssen (Ed.), *Modeling in DataBase Management Systems* (pp. 201-220). North Hollan.

Hull, R. & King, R. (1987). Semantic database modeling: survey, applications, and research issues. *ACM Computing Surveys (CSUR)*, **19**(3), 201-260.

Kaburlasos, V.G. (2006). *Towards a unified modeling and knowledge-representation based on lattice theory*. Springer.

Kent, W. (1991). A rigorous model of object references, identity and existence. *Journal of Object-Oriented Programming*, **4**(3), 28-38.

Khoshafian, S.N. & Copeland, G.P. (1986). Object identity. In *Proc. OOPSLA'86*, *ACM SIGPLAN Notices*, **21**(11), 406-416.

Kim, W., Garza, J.F., Ballou, N. & Woelk, D. (1990). Architecture of the ORION next-generation database system. *IEEE TKDE*, **2**(1), 109-124

Kimball, R. & Strehlo, K. (1994, June). What's wrong with SQL. *Datamation*.

Lieberman, H. (1986). Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. OOPSLA'86*, *ACM SIGPLAN Notices*, **21**(11), 214-223.

Pardillo, J., Mazón, J.-N. & Trujillo, J. (2008). Bridging the semantic gap in OLAP models: Platform independent queries. In *Proc. DOLAP'08* (pp. 89-96).

Pedersen, T.B. (2009). Multidimensional modeling. In L. Liu & M.T. Özsu (Eds.), *Encyclopedia of Database Systems* (pp. 1777-1784). Springer, NY.

Raymond, D. (1996). *Partial order databases.* Ph.D. Thesis, University of Waterloo, Canada.

Rizzi, S., Abelló, A., Lechtenbörger, J. & Trujillo, J. (2006). Research in data warehouse modeling and design: dead or alive? In *Proc. DOLAP'06* (pp. 3-10).

Savinov, A. (2014a). Concept-oriented query language. In J. Wang (Ed.), *Encyclopedia of Business Analytics and Optimization*. IGI Global.

Savinov, A. (2014b). Concept-oriented model. In J. Wang (Ed.), *Encyclopedia of Business Analytics and Optimization*. IGI Global.

Savinov, A. (2014c). Concept-Oriented Programming. In Mehdi Khosrow-Pour (Ed.), *Encyclopedia of Information Science and Technology, 3rd Edition*. IGI Global.

Savinov, A. (2012a). Concept-oriented model: Classes, hierarchies and references revisited. *Journal of Emerging Trends in Computing and Information Sciences,* **3**(4), 456-470.

Savinov, A. (2012b). Inference in hierarchical multidimensional space. In *Proc. International Conference on Data Technologies and Applications (DATA 2012)* (pp. 70-76).

Savinov, A. (2012c). Concept-oriented programming: Classes and inheritance revisited. In *Proc. 7th International Conference on Software Paradigm Trends (ICSOFT 2012)* (pp. 381-387).

Savinov, A. (2012d). References and arrow notation instead of join operation in query languages. *Computer Science Journal of Moldova (CSJM),* **20**(3), 313-333.

Savinov, A. (2011a). Concept-oriented query language for data modeling and analysis. In L. Yan and Z. Ma (Eds.), *Advanced Database Query Systems: Techniques, Applications and Technologies* (pp. 85-101). IGI Global.

Savinov, A. (2011b). Concept-oriented model: Extending objects with identity, hierarchies and semantics. *Computer Science Journal of Moldova (CSJM),* **19**(3), 254-287.

Savinov, A. (2009). Concept-oriented model. In V.E. Ferraggine, J.H. Doorn, & L.C. Rivero (Eds.), *Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends* (2nd ed., pp. 171-180). IGI Global.

Savinov, A. (2008). Concepts and concept-oriented programming. *Journal of Object Technology (JOT), 7*(3), 91-106.

Savinov, A. (2006a). Grouping and aggregation in the concept-oriented data model. In *Proc. 21st Annual ACM Symposium on Applied Computing (SAC'06)* (pp. 482-486).

Savinov, A. (2006b). Query by constraint propagation in the concept-oriented data model. *Computer Science Journal of Moldova (CSJM)*, **14**(2), 219-238.

Savinov, A. (2005a). Hierarchical multidimensional modelling in the concept-oriented data model. *Proc. 3rd international conference on Concept Lattices and Their Applications (CLA'05)* (pp. 123-134).

Savinov, A. (2005b). Logical navigation in the concept-oriented data model. *Journal of Conceptual Modeling, 36*. Retrieved December 5, 2012, from http://conceptoriented.org/savinov/publicat/jcm_05.pdf

Sibley, E.H., & Kerschberg, L. (1977). Data architecture and data model considerations. In *Proceedings of the AFIPS Joint Computer Conferences* (pp. 85-96).

Smith, J.M. & Smith, D.C.P. (1977). Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems (TODS)*, **2**(2), 105-133.

Stonebraker, M., Rowe, L., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P. & Beech, D. (1990). Third generation database system manifesto. *ACM SIGMOD Record*, **19**(3).

Wieringa, R. & de Jonge, W. (1995). Object identifiers, keys, and surrogates - object identifiers revisited. *Theory and Practice of Object Systems*, **1**(2), 101-114.

Zaniolo, C. (1984). Database relations with null values. *Journal of Computer and System Sciences*, **28**(1), 142-166.