VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.

http://www.cisjournal.org

Concept-Oriented Model: Classes, Hierarchies and References Revisited

Alexandr Savinov

SAP Research, Chemnitzerstr. 48, 01187 Dresden, Germany

http://conceptoriented.org/savinov

ABSTRACT

We present the concept-oriented model (COM) and demonstrate how its three main structural principles — duality, inclusion and partial order — naturally account for various typical data modeling issues. We argue that elements should be modeled as identity-entity couples and describe how a novel data modeling construct, called concept, can be used to model simultaneously two orthogonal branches: identity modeling and entity modeling. We show that it is enough to have one relation, called inclusion, to model value extension, hierarchical address spaces (via reference extension), inheritance and containment. We also demonstrate how partial order relation represented by references can be used for modeling multidimensional schemas, containment and domain-specific relationships.

Keywords: *Data modeling, Conceptual models, Unified models, Data semantics, Data types, Multidimensional models, Inheritance, Partial order, References*

1. INTRODUCTION

1.1. Motivation and Major Goals

One of the primary concerns in data modeling is data organization. Currently there exist many approaches to data organization relying on different structural principles. They implement various patterns of thought which are most appropriate for the corresponding application domain. Existing models use many different definitions and interpretations of semantic relationships which often result in quite ambiguous, incomplete and inconsistent conceptual specifications. Databases are split into many specialized solutions like transactional, analytical and semantic systems which use different models and query languages. Many of these problems are due to the existence of a number of deeply rooted incongruities, incompatibilities and asymmetries between different views of data which are shortly outlined below.

Value vs. object modeling. Values (data passed by-copy) and objects (data passed by-reference) have always been considered two separate branches: either we model value types or we model object types. A typical example is the object-relational model [17, 36, 37] where user-defined types are modeled separately from relations. Our goal here is to unify these two branches by using only one construct for defining types and only one kind of domains.

Identity vs. entity modeling. There exist numerous studies [16, 39, 15, 10] highlighting identities as an essential part of data and programming models. Nonetheless, almost all existing data models have a strong bias towards modeling entities while identities (references, addresses, surrogates, OIDs) are considered secondary elements being modeled by means of entities. An extreme view is that identities belong to a physical level and should not be modeled at all (at least at the same level as entities). We would like to eliminate this asymmetry by making identities and entities equally important parts of a data element both being in the focus of data modeling and modeled at the same level.

Instance-based vs. set-based modeling. Most models principally separate the notions of individual elements and sets of elements. 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 (of tuples) which however is not treated and operated like a normal data element (tuple or object). The goal here is to eliminate this difference so that any instance is inherently a set (of other instances) and hence sets are also normal instances. Another goal is to put any element in space (domain, context or scope) where it exists because a thing *in vacuo* outside of any space is considered nonsense. Data management is then reduced to maintaining set membership while other interpretations are derived from this relation. Ideally, all operations with data should be reduced to only adding an element to a set and removing an element from a set (where sets are other elements). In particular, properties also should be interpreted from the point of view of set membership relation rather than an independent mechanism.

Transactional vs. analytical modeling. Assumptions and techniques behind analytical models are different from those behind transactional models. The primary notion in analytical models is that of dimension [21, 22] while transactional models are aimed mainly at modeling entities and relationships. Most conventional database systems provide very limited analytical functions while analytical systems are not intended for transactional processing at all. One negative consequence of this incompatibility is that one and the same data is modeled and managed two times: in a transactional database and in a data warehouse [8, 12, 18]. The goal of COM here is to develop a data model which could be used for both purposes without any adaptation or tuning. In particular, dimensions and dimensionality (degrees of freedom) should be first-class

©2009-2012 CIS Journal. All rights reserved.

notions of the model rather than something auxiliary being added separately at later stages.

Conceptual vs. logical modeling (semantic gap). The main purpose of conceptual models is to provide richer mechanisms and constructs representing complex application-specific concepts and relationships with the goal "to respond to queries and other transactions in a more intelligent manner" [7]. Yet, the translation procedure can be quite ambiguous and non-trivial because of the qualitative differences between the two levels of representation termed as the semantic gap [24, 20]. This frequently leads to the need in two separate models: one semantic model and one logical model. In this context, the goal is to make semantics integral part of the logical data model so that a database can directly use it for reasoning about data and maintaining its consistency.

Entity vs. relationship modeling. Relationships may have properties, identities and their own relationships while entities may well be interpreted as relationships between other entities. Therefore it can be quite difficult to decide whether a domain concept should be represented as an entity or relationship, and this ambiguity can lead to different representations of the same data created by different designers. Removing these differences between entities and relationship by unifying their treatment would significantly simplify data modeling.

Attributes vs. relationships. Both attributes and relationships are used to define model structure and connectivity. Existing conceptual models provide both mechanisms and therefore it can be difficult to decide which of them to use. For example, if a person lives in some country then this can be represented either by a field livesIn or a relationship livesIn. Our goal here is to unify these mechanisms by using a more general notion of reference.

Data modeling vs. programming. This problem is frequently associated with the object-relational impedance mismatch [1, 9] but it has a wider scope because data is modeled and manipulated differently in programming and database areas. In particular, we would like to have the same type system suitable for both programming and databases.

This paper presents the concept-oriented model (COM) which is a means of addressing the above problems. COM is a unified general purpose model which reduces a wide variety of existing data modeling methods to a few novel structural principles. Its main goal is to radically simplify data modeling by covering a large number of existing techniques and patterns. It also aims at reducing the number of primary data modeling constructs and increasing semantic integrity.

1.2. Model Structure and Principles

The general structure of the model is shown in Fig. 1 and its major principles are shortly described below.

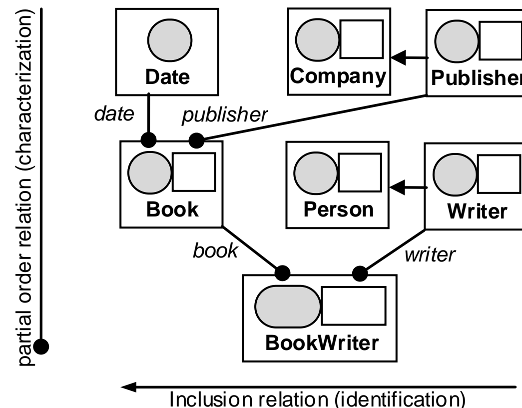

Figure 1. Structure of the concept-oriented model

Duality principle. Probably the most wide spread data modeling mechanism is to defining a new type by *combining* several existing data types. For example, a book is a combination of a publisher and a publication date. This idea is so general, simple and natural that it is supported by almost all data models where existing type definitions can be used as attribute types within other type definitions. It is so deeply embedded in our thinking that it is difficult to imagine how and for what reason it could be changed. Yet, we argue that defining a new type as a combination of other types is not enough for a good data model for at least two major reasons: (i) it is not possible to represent the distinction between by-value and by-reference semantics, and (ii) it is not possible to model references which provide a basic connectivity mechanism.

COM revisits this fundamental pattern by postulating in its duality principle that *any element has two parts: identity and entity.* Very informally, elements in COM can be thought of as complex numbers in mathematics which also have two constituents but are modeled and manipulated as one whole. Identity is passed by-value and entity is passed by-reference where reference is the corresponding identity. In Fig. 1, identities are shown as grey rounded rectangles coupled with entities which are shown as white rectangles. To model such identity-entity couples COM introduces a novel data modeling construct, called *concept* (hence the name of the model) which is defined as a couple of two classes: one *identity class* and one *entity class*. Instances of identity classes are values which are passed by-value while instances of entity classes are objects which are passed by-reference (that is, by means of identities). For example, a book could be modeled as a couple of its isbn which is its identity and its title which is part of its entity. Storing a reference to a book means storing its isbn as a value. If an element has no entity part then it is a value (like Date in Fig. 1) so that we can easily model value domains. If an element has no identity explicitly defined then it is a conventional entity or object represented by some kind of primitive reference (inherited from the root concept).

VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.
http://www.cisjournal.org

Inclusion principle. Hierarchies are supported by most data models but they have different purposes and interpretations like containment and inheritance. In the inclusion principle, COM postulates that *all elements (except for one root element) exist in a hierarchy where identities of child elements are defined with respect to the identity of the parent element*. Inclusion hierarchy spreads horizontally in Fig. 1 so that an element is a set of its child elements positioned to the right. For example, concept Publisher is included in concept Company which means that one company may have many publishers within it (say, many publishing departments) and each publisher instance is identified with respect to the company. Elements of this hierarchy are uniquely identified by a sequence of identity segments starting from the root (leftmost concept on diagrams) and ending with the represented element. Such complex identities are analogous to conventional postal addresses. An important property of inclusion relation is that it integrates identification, containment, inheritance and generalization relations. Including an element into a parent element means to identify it with respect to the parent, to inherit from the parent and to be more specific than the parent. For example, concept Writer is included in Person which means that Writer inherits from Person and can use its properties.

Order principle. This principle postulates that *all elements are partially ordered*. Partial order is represented by references by assuming that an element references its greater elements. This structure is orthogonal to the structure of inclusion relation and spreads vertically in Fig. 1. Concepts are used to model partially ordered structure of elements by assuming that field types specify greater concepts. For example, a Book is a combination of its publishing Date and Publisher. Therefore, the latter two concepts are greater than the Book concept (greater concepts are positioned higher than lesser concepts in diagrams). Partial order is used in many models as a natural constraint imposed on existing relations. What is new in COM is that instead of introducing some specific mechanisms and then imposing partial order as a constraint we postulate that a model *is* a partially ordered set which is later interpreted from various points of view by using different assumptions about the nature of data. Data modeling is then significantly simplified because it is reduced to partially ordering concepts by defining their field types. The main benefit is that partial order "seems to fulfill a basic requirement of a general-purpose data model: wide applicability" [23], that is, many conventional data modeling mechanisms and patterns can be unified and explained in terms of this formal setting.

Recently, a number of papers have been published [26, 27, 28, 32, 33, 34] which describe either preliminary results or specific mechanisms of COM with the focus on query and analysis tasks. This paper focuses mainly on conceptual data modeling, data semantics and type modeling. We describe principles of the concept-oriented model from the point of view of existing approaches to conceptual data modeling. Each section describes one data modeling task and demonstrates how it can be solved using COM principles. The next three sections of the paper are devoted to describing the three principles of COM (duality, inclusion and partial order) and the last section makes concluding remarks.

2. DUALITY — CLASSES REVISITED

2.1. By-Value or By-Reference? Both

The most wide-spread approach to defining a new type consists in *combining* several already existing types. For example, a class is a combination of fields and a relation is a combination of attributes. The problem is that such definitions do not provide any indication whether they describe values (passed by-value) or objects (passed by-reference). Yet, without this information the type cannot be used because the size of its instances cannot be determined. If it is a value then the size is the sum of all field sizes, and if it is an object then its size is the size of its reference. For example, if type A is defined as a combination of types B and C, TYPE A {B f1; C f1}, then this definition is not complete because we do not know if instances of A will contain references to B and C or the values of types B and C. In most models, this ambiguity is resolved by using implicit assumptions about the semantics of their types (for instance, OOP assumes that class instances are passed by-reference) or by introducing a mechanism for specifying it for each individual element or context (for instance, a field could be marked by the keyword by-ref or by-val). Note that this problem exists also in conceptual models. For example, UML uses classes to describe what is supposed to be passed by-reference and data types to describe values.

The main limitation of this classical approach is that it provides only two extreme options when choosing between by-value and by-reference semantics: either primitive references for representing objects or domain-specific values. In particular, domain-specific structure can be specified either for values or for objects but not for both. If we want to pass some data by-value then it is not possible to interpret this value as a reference which represents some object. And if we want to pass some data by-reference then this reference can be only of primitive type and there is no way to specify arbitrary structure for it. In other words, it is not possible to specify that one part of an element has to be passed by-value by representing the second part which is an object passed by-reference. In the case of the relational [6] and object-relational [17, 36, 37] models, this classical approach leads to the limitation that relations cannot be used as attribute types:

TYPE MyValueType // Instances are values
  // Relation as a type not possible
  myField ~~AS MyRelationType~~
END TYPE

RELATION MyRelationType
  // Relation as a type not possible
  myAttribute ~~AS MyRelationType~~
END RELATION

In the case of object data models [2, 3, 10], the limitation is that fields will always contain primitive

VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.
http://www.cisjournal.org

values (OID) without a possibility to specify their structure:

```
class MyObjectType
  MyClass myField // No complex references
```

If we want to have an arbitrary (domain-specific) value stored in a field then it can be done by defining a value type but in this case we lose the possibility to reference an object:

```
class MyObjectType
  // Value is not a reference
  MyValueType myField
```

Ideally, we would like to have a single construct for defining types without the separation between value types, object types, relation types or entity types. But at the same time, we would like to have a possibility to specify what part of the structure is represented by-value and what part is represented by-reference.

It is a quite challenging problem which touches the fundamentals of data modeling. To solve it, COM revisits the notion of type by proposing to distinguish between, and explicitly specify, the type constituents to be passed by-value and by-reference. More specifically, COM introduces a new data modeling construct, called *concept*, defined as follows:

Definition [Concept]. Concept is a couple consisting of one identity class and one entity class, where instances of the identity class are passed by-value and instances of the entity class are passed by-reference using the identity.

The concept construct is compatible with the classical approach by producing two particular cases. If entity class is empty then the concept describes a value type because it consists of only an identity class, instances of which are passed by-value with absent referent. And if identity class is empty then it is equivalent to an object type, instances of which are passed by-reference. Strictly speaking, elements may not have empty identity because they must be somehow identified (no identity means non-existence). Therefore, empty identity class means that the identity will be inherited from the parent concept (see Section 3).

For example, colors could be described as either values or objects:

```
CONCEPT ColorValue //Instances are values
  IDENTITY
    INTEGER red, green, blue
  ENTITY // Empty

CONCEPT ColorObject //Instances are objects
  IDENTITY // Empty
  ENTITY
    INTEGER red, green, blue
```

However, the main use of concepts is in defining both constituents, which is not possible in existing models. For example, traditional methods do not allow us to define colors as elements identified by name (passed by-value) representing three constituents passed by-reference:

```
CONCEPT Color
  IDENTITY
    CHAR(10) name
  ENTITY
    INTEGER red, green, blue
```

Although any concept has two constituents, this internal division is not visible when it is used. Once a concept has been defined we can forget about its internal structure and use it as a conventional type. There is no difference between concepts and classes when using them for defining element types. The difference is that concept-typed elements store identity and this value references the entity (both having arbitrary structure defined in this concept). For example, the first two variables below are value-typed and object-typed, while the third variable is concept-typed:

```
ColorValue val // Stores a value
ColorObject obj // Primitive reference
Color field // CHAR(10) represents object
```

One of the most important benefits of the concept-oriented approach is that it generalizes and simplifies the object-relational model (as well as other approaches where attributes take values from a domain). These conventional approaches consider value domain modeling as a separate branch (Fig. 2a). In particular, if relations need to be modeled then it is done using an orthogonal approach. In object models classes are used to model objects (Fig. 2b) and values are modeled separately or only primitive values are used. The main benefit of concepts is that they are used for modeling both parts (Fig. 2c) by using only one kind of domains consisting of identity-entity couples (rather than value domains and relations). Therefore, it is enough to specify attribute types as concepts and then depending on the concept internal structure it will store either values or represent tuples in other relations. The duality principle has also many other advantages, particularly, for modeling references which are described in the next sections.

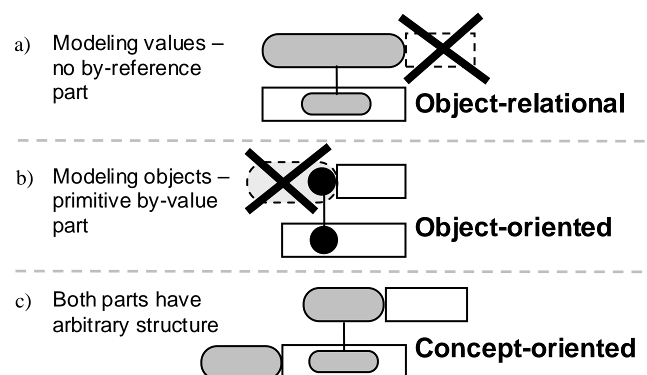

Figure 2. An element is an identity-entity couple

2.2. Modeling References

An entity is a thing which can be uniquely identified and has an independent existence. But what does it mean

©2009-2012 CIS Journal. All rights reserved.

for an entity to exist? Most models recognize the importance of identities as a means for *representing* entities: Yet, they do not answer the question what the identity is and how it should be modeled. There is an old belief that it is entity that should be in the focus of data modeling while identities (references) simply serve entities and should not be explicitly modeled. Therefore, the role of identities is still underestimated and their support is rather weak. The standard approach to identifying entities consists in selecting special attributes which are required to be unique and then are used to distinguish the entities. In this sense, the problem of identification is reduced to the problem of unique characterization. COM follows a different approach where identities exist separately from the represented entity which is closer to how objects are represented in the object-oriented model.

In contrast to this dominating view, we argue that references not only should be integral part of the model but they should also have the same rights as entities. Identities are supposed to be as important as entities and a good data model should provide means for modeling arbitrary user-defined types of references. It is equally important to be able to directly model both identities and entities by retaining the differences between them. Identities in COM manifest the fact of the *existence* of a thing and modeling identities means describing how things exist. A thing is assumed to exist if it has an identity and if a thing does not have an identity then it is assumed to be non-existing. One of the crucial points in our argumentation is that references are normal values (which provide access to the values stored indirectly within objects). In other words, identities *are* data, that is, they are precisely what is transferred and what is stored. Therefore, if values are supposed to be integral part of the model then it is quite natural that there have to be means for modeling identities which are also values. Another point is that identities are an important part of the problem domain and hence a model should provide means for their description. For example, there could be a model describing only identities (an identification schema) without entities and it is much more convenient to use dedicated means for their description rather than by adopting entities for that purpose.

Currently there exist two major approaches to modeling identities:

- [Primitive references] Identities are automatically provided by the environment for all entities and cannot be modeled
- [Identifier keys] A subset of entity attributes is used for identification purposes

The first approach (Fig. 3a) means that domain-specific references are completely excluded from the model. The problem here is that it is not possible to *directly* model identities existing in the problem domain. For example, postal addresses, bank accounts, passport numbers, insurance codes and other domain-specific identities have to be modeled as entities represented by primitive references. Yet, the correct approach would be to model them directly as identities representing the corresponding entities.

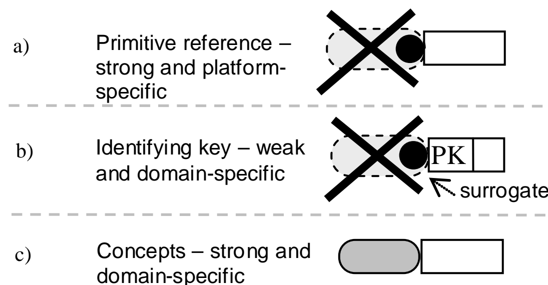

Figure 3. Both identity and entity have arbitrary structure

The second approach (Fig. 3b) solves the problem of domain-specific references by marking some entity attributes as keys. Yet, the problem with this approach is that even if some entity attributes are marked as keys they still belong to the entity. In particular, it is not possible to model references as value domains and this approach can be viewed as a pattern (with some advantages and disadvantages). For example, if banks are identified by their BIC (Bank Identifier Code) then it can be defined as an attribute marked as a key:

```
class Bank
  key CHAR(11) bic // Key attribute
  CHAR(64) name // Normal attribute
  Bank b // Primitive reference - not a key
```

Importantly, instances of this class will have three attributes (not two) and variables of this class will still store a primitive reference rather than a key. Thus it can be viewed as a simulation of true domain-specific identities.

Concepts (Fig. 3c) provide a principled solution to this problem because identities and entities are defined as two symmetric constituents of a data type. For example, the previous example can be modeled using the following concept:

```
CONCEPT Bank
 IDENTITY
   // Identity attribute is a value
   CHAR(11) bic
 ENTITY
   CHAR(64) name // Entity attribute
   Bank bank // Stores BIC
```

In contrast to using identifier keys, entities of this concept will have only two attributes while the first attribute (bic) is treated as a value which is not stored within this same entity. The last line looks like a conventional field definition. However, a reference stored in this field (bank) has a domain-specific structure described in the identity class of the Bank concept.

A concept can be thought of as a description of custom memory where identities are used to access cell contents. The difference from conventional hardware memory is

©2009-2012 CIS Journal. All rights reserved.

that both addresses and cells have arbitrary structure. Importantly, cells do not store their addresses, that is, memory is an array of (addressable) cells rather than a two-column table of address-cell pairs.

Identifier keys are also used in the relational model [6] where primary key (PK) attributes are intended for identifying tuples and foreign key attributes (FK) are used to reference other tuples. Although this approach allows for modeling domain-specific identities, it has a number of quite serious drawbacks described below.

Weak type information. FK attributes (taken as whole) do not declare the type of the entity they represent. Instead, each FK attribute has a primitive (or user-defined) type only. For example, if an attribute is intended to represent a bank entity then it is declared as having type CHAR(11):

```
bank AS CHAR(11) // True type is absent
```

This definition is misleading because both the data modeler and the database system are completely unaware that this attribute represents a bank. Therefore, it will be manipulated as a character string while its true type is Bank and has to be manipulated like Bank. Of course, it is possible to restore the true type from the FK declaration. However, FKs are not part of the type system and are not mandatory part of schemas. Therefore, they may well be absent and then it is not possible to restore true attribute types at all. But even if FK declarations are present and the true attribute types can be restored then the question is what do we need attribute types for? FK attribute types in this case (like CHAR(11) in the above example) are redundant because they can be restored from the corresponding PK. For example, if we know (from FK) that an attribute represents a Bank then from the Bank definition (from its PK) we can derive that this attribute has to store a value of type CHAR(11). Thus the relational schema duplicates type information which makes it error-prone and difficult to maintain. COM does not have this drawback because concepts incorporate both the type of reference and the type of the represented entity in one construct.

PK structure is a cross-cutting concern. Assume that a primary key PK1 is defined as a set of attributes used for identification purposes. If PK2 references PK1 via FK2 then all attributes of PK1 are declared again in FK2, that is, the structure of PK1 is repeated in FK2 (which is part of PK2). If the third primary key PK3 declares FK3 representing PK2 then this FK3 will be declared as a set of all attributes from PK2 and PK1. Thus the structure of primary keys is repeatedly defined each time this key is used in a foreign key in nested manner. It is a typical cross-cutting concern where the same piece of code or type information cross-cuts the whole program or schema. Maintaining such schemas and queries can be a quite difficult task because any small change in a PK structure will have to be propagated all over the schema and queries. COM schema does not have this drawback because all necessary type information is declared only once and then the name of the concept is used to refer to it.

Join does not reflect its purpose. Join operation has several major purposes which cannot be distinguished in its syntax. For example, if an SQL query specifies some join condition like WHERE A.id=B.id then it says almost nothing about its real intention. The meaning of queries is not directly expressed in the structure of operations. In particular, we do not know whether table A references table B or vice versa (or maybe it is not about referencing at all). Concepts allow us to overcome this complexity because one concept describes one reference: identity class describes the structure of reference while entity part described the structure of the referenced part. As a result, all the underlying mechanics of references like join conditions is not explicitly expressed in queries but concepts have the necessary information for translating such queries into low level database operations.

3. INCLUSION — HIERARCHIES REVISITED

3.1. Modeling Hierarchical Identities and Value Extension

The main advantage of using domain-specific identities is that they allow us to directly model arbitrary address spaces which is difficult or impossible in other models. Yet, this approach is still limited because all elements of one type exist in one flat space and there is no way to describe structural addresses. For example, it is difficult to model conventional postal addresses where one segment like city is a relative address specified with respect to its parent segment like country. We also cannot model bank accounts which are defined relatively to their bank. In this case bank code is a parent segment and account number is a child segment of the address. The whole address consisting of several segments is an element in a hierarchical address space. One parent segment has many child address segments so a parent can be viewed as a set of its children. Although the notions of scope, space or context are considered in many models [15], they are provided as an additional mechanism or relationship while our goal is to introduce structural identities as an integral and primary part of the model.

In the previous section we assumed that identity is a value. To solve the problem of hierarchical addresses we additionally assume that an *extension* of a value is a relative or local address with respect to the original (base) value. The operator of extension is denoted by colon. For example, if 'DE' is a value identifying a country then 'DE':'Dresden' is an extended value which identifies a city within this country identified by the value 'Dresden'. If simple identities manifest the fact of existence then complex identifies manifest the fact of *existence in space* which means that an extended element exists in the space identified by its parent base segment.

To model value extension and relative addresses COM introduces a new relation between concepts called *inclusion*, which specifies a parent concept this concept is included in. The parent concept is also called a *super-concept* while a child concept is referred to as a *sub-concept*. Elements are still identified by their identities but

Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.
http://www.cisjournal.org

now these identities are extensions of the parent identities which in turn extend their parent identity and so on up to the root which represents the whole space of elements or the universe of discourse. Thus a fully-specified identity of an element is a value which consists of several segments starting from the root and ending with the identity of this element.

For example, if bank accounts are identified with respect to their bank (account without a bank is meaningless) then concept Account has to be included in concept Bank:

```
CONCEPT Account IN Bank // Exists in bank
 IDENTITY
  // Extends identity of Bank
  CHAR(10) accNo
 ENTITY
  DOUBLE balance // See next section
```

If accounts are supposed to have savings accounts (savings account without some main account is meaningless) then this concept is included in the parent Account concept:

```
CONCEPT Savings IN Account // In account
 IDENTITY
  // Two digits extend account number
  CHAR(2) savAccNo
 ENTITY
  DOUBLE savings // See next section
```

As a result, a bank consists of a number of accounts and each account is a set of savings accounts. Thus all elements exist in a hierarchical address space where they are distinguished by their complex identifies which are multi-segment values. Like any value, complex identities can passed and stored but they do not have their own references and therefore cannot be shared (they can only be copied). Inclusion in COM is similar to XML structure where parent elements contain child elements directly by-value.

A variable of a concept may store elements of its child concepts. For example, a value of concept Savings can be assigned to a variable of concept Account:

```
Savings savings = getSavings();
Account account = savings;
```

This possibility to have additional segments in references is a concept-oriented analogue of polymorphism. If concepts have behavior then it results in even more interesting mechanisms like reverse overriding in concept-oriented programming (COP) [25] where parent method override child methods [29, 30, 31].

3.2. Modeling Hierarchical Entities and Entity Extension

One problem of conventional inheritance is asymmetry between classes and their instances: classes exist in a hierarchy where parents are shared among all their child classes while instances still exist in a flat space where parents are not shared and each child has its own copy of the parent data. It is not possible to reuse the same parent instance by creating several different extensions in its context. We can model hierarchies of classes but are not able to produce the corresponding hierarchy of instances. Since instance hierarchies are as important as class hierarchies, these two structures are modeled using different means: class hierarchies are modeled using inheritance while instance hierarchies are modeled using some kind of containment relationship. The difficulty of the existing methods is that they introduce many independent types of relations for expressing similar structures. There are also approaches where classes are not used [19] so that instances exist in a hierarchy [4] and inheritance is implemented via delegation [35]. Our goal is to use only one inclusion relation for representing many different semantics like address hierarchies, instance hierarchies, inheritance, containment and generalization-specialization.

As described in the previous section, inclusion is equivalent to value extension when applied to identities (interpreted as a relative or local address). But what about entities? Here concepts are cardinally different from what is expected from classes and classical extension. COM assumes that parent entities are shared parts of their children and one parent entity may have many child entities. For example, all account instances created within one bank see one and the same bank name which is an entity field in the Bank concept. And all savings accounts created within one main account see the same main balance which is an entity field in the Account concept. Thus entities also exist in a hierarchical space where they are identified by the corresponding hierarchical addresses. Note that this is only possible because of the presence of identities which serve as local addresses for entities in the hierarchy. And this is why having instance (entity) hierarchies is not possible by using conventional classes where child instances cannot be distinguished within their parent. Thus COM inclusion eliminates the asymmetry between classes and instances so that a concept hierarchy directly models instance hierarchies.

Since an element may have many children, inclusion can be interpreted as a containment relation: any instance is a *set* of its child instances and any instance is a *member* within its parent instance so that the whole approach is inherently set-based. Since identities cannot be (easily) changed, elements cannot change their parent. It is a natural consequence of containment by-value where an element is created and always exists as a whole within one parent. An alternative containment by-reference is described in Section 4.2.

Although modeling instance hierarchies and containment is a very important issue, we still would like to have a mechanism for modeling conventional inheritance. In COM, classical inheritance is a particular case of inclusion where a child concept (extension) has empty identity class. Such concepts are equivalent to conventional classes because their instances cannot be distinguished in the context of the parent instance. As a consequence, only one child instance can be created which is considered an extension of the parent. For example, if

©2009-2012 CIS Journal. All rights reserved.

we need to describe some very specific kind of account with an additional entity field then we can do it as follows:

```
CONCEPT GoldAccount IN Account
 IDENTITY // No identity => inheritance
 ENTITY
  DOUBLE interestRate // Entity extension
```

A new instance of the parent concept is created for each instance of this concept. Therefore, the interestRate field can be thought of as directly extending the parent field (balance in this example).

Inclusion can be illustrated using the following example. Assume that there are many rooms identified as 'R1', 'R2', 'R3' and existing in one building which is identified as 'B1'. All rooms exist in the building by-value and cannot be easily moved to another building. Therefore, it does make sense to identify them with respect to the building, for example, room 'B1': 'R2' is the second room in the first building. Inheritance is a special case where a building is known to have only one room (say, a hall). Such rooms do not need a separate identity just because they can be uniquely identified using the building they are in (we say that a child inherits or reuses the parent identity). Such a room is viewed as an extension of the building because it simply adds some more specific attributes like the number of seats. Thus COM generalizes classical inheritance and provides a novel view on this relation by using the following principle: *to be included in a container means to inherit its properties*. In particular, members of a set automatically inherit properties of this set. This significantly simplifies data modeling because one relation is used to describe containment and inheritance.

Another property of COM inclusion is that child elements are treated as more specific than their parents and hence inclusion can be used to represent general-specific relation. In terms of sets this means that *members are more specific elements than the set they are in*. To produce an element of a set we extend this set (by adding addition identity segments) and simultaneously more specific elements are produced.

This approach can be successfully applied to relational modeling where inheritance was shown to be not very appropriate. Relation types can be declared as concepts with identities implemented as primary keys (but manipulated according the concept-oriented treatment of true identities). If a relation type needs to be extended then it is done by means of inclusion relation between the corresponding concepts. The extended relation will contain only instances of the child concept and there can be many such instances belonging to one parent instance stored in the parent relation. In the case of inheritance, identity of the extended relation type is empty and each parent tuple has maximum one child tuple. Note that if a relation type defines only identity class (with empty entity) then it is equivalent to defining a domain (value type) as it is done in the object-relational model (see Section 2). In this way we can significantly diminish the differences between relational and object-oriented modeling because concepts provide a common mechanism for modeling simultaneously value domains and relation types.

4. PARTIAL ORDER — REFERENCES REVISITED

4.1. References for Multidimensional Modeling

A variety of approaches and techniques have been proposed for representing multidimensional data [21, 22]. Yet, one of their main problems is that these models are not intended for transactional processing and can be viewed more as models of analysis. What is worse, the analysis scenarios described by such models in great extent reflect the needs of concrete applications because defining cubes, dimensions and measures depend on what an application needs. Therefore, a data model is split into a transactional part which is application-independent and an analytical part which is application-oriented. This deeply rooted incongruity can be termed as a transactional-analytical impedance mismatch and the need to have two models for the same data results in numerous problems at all stages of the enterprise data management system life-cycle [8, 12, 18]. Currently there exist systems which serve as common storage for both transactional and analytical data. Yet, data modeling is still performed separately, that is, first a transactional model is created (possibly with a separate conceptual model) and then a multidimensional analytical model is provided which relies on the transactional model.

COM eliminates differences between the two types of modeling so that one and the same model can be used for both transactional and analytical applications. The idea is based on the COM order principle which postulates that all elements in the model are partially ordered. Further, partial order is represented by references by assuming that referenced elements are greater than the referencing element. An important property of partial order is that it can be interpreted as a multidimensional hierarchical space where greater elements are interpreted as *coordinates* with respect to their lesser elements which are interpreted as *points*. For example, if a book element references a publisher then the book is interpreted as a point while the publisher is one of its coordinates.

Concepts are also partially ordered by assuming that field types specify greater concepts. Each concept describes a multidimensional space with the fields interpreted as dimensions (it is precisely why concept fields in COM are referred to as dimensions). For example (Fig. 1), if books are characterized by publishers then this means that the Publisher concept is greater than the Book concept:

```
CONCEPT Publisher ... // Greater concept

CONCEPT Book // Lesser concept
 ENTITY
  // Type is a greater concept
  Publisher publisher
```

©2009-2012 CIS Journal. All rights reserved.

According to the multidimensional interpretation, the Book concept describes points and the Publisher concept describes coordinates.

Another example is shown in Fig. 4a where concept BookWriter has two fields of concepts Book and Writer. The Book concept has two fields referencing one Publisher and a Date of publishing. (In diagrams, greater concepts are positioned above their lesser concepts.) BookWriter is a lesser concept and hence it is positioned below Book and Writer which are greater concepts. However, the Book concept has also two fields of type Date and Publisher which therefore are greater concepts positioned higher than the Book concept. The multidimensional space for this schema is shown in Fig. 4b. BookWriter describes a 2-dimensional space because it has two greater concepts, Book and Writer. However, Book is also a 2-dimensional space with its own two dimensions Date and Publisher. The total model dimensionality is equal to the number of dimension paths from bottom to top concept. The model in Fig. 4 has dimensionality 3 which means that data element of concept BookWriter can be varied along three dimensions.

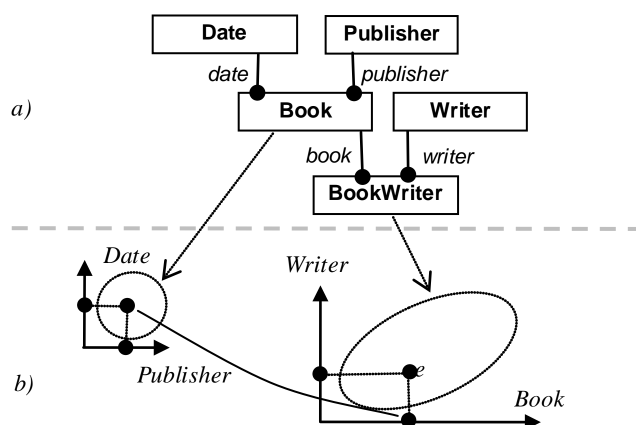

Figure 4. Partial order interpreted as a multidimensional space

This multidimensional interpretation is supported by operations of *projection* and *de-projection* denoted by right and left arrows, respectively. Projection means moving up in the partially ordered set to a greater concept. When applied to a set of elements (interpreted as points) it returns a set of their coordinates along the specified axis. A set of elements of some concept is denoted by this concept name written in parentheses. For example, given a set of books we can find the related publishers by projecting along the publisher dimension:

```
(Book | date > '01.01.2005')
  -> publisher -> (Publisher)
```

De-projection is the opposite operation which means moving down to a lesser concept. When applied to a set of elements (interpreted as coordinates) it returns a set of points which take them. For example, given a set of publishers we can de-project them down and find all their books:

```
(Publisher | name = 'XYZ')
  <- publisher <- (Book)
```

These operations serve as a basis for data analysis in COM. The general idea is that constraints are specified in some parts of the schema and then propagated to another part of the schema using a zig-zag dimension paths composed of projections and de-projections [27, 33]. For example, we could easily find all writers of a publisher by applying two de-projections followed by projection:

```
(Publisher | name = 'XYZ')
   <- publisher <- (Book)
   <- book <- (BookWriter)
   -> writer -> (Writer)
```

One novel feature of this approach is that we can carry out inference on data by automatically propagating constraints to the target [28, 33]. In this case the above query is written even simpler without specifying a constraint propagation path:

```
(Publisher | name = 'XYZ') <-*-> (Writer)
```

Here <-*-> is inference operator which combines de-projection and projection.

An advantage of this approach is that it does not add complexity to the model but rather interprets partial order in terms of dimensions, points, coordinates and other notions used in multidimensional data modeling and analysis. From this point of view, adding a new field to a concept means adding a new dimension to the model. Data is thought of as originally existing in a multidimensional space so that it is always possible to say what coordinates this element has and how many dimensions this schema has. One and the same model can be used for modeling both transactional data and analytical data by eliminating the transactional-analytical impedance mismatch. COM is also more flexible than standard OLAP models because it does not rely on predefined cubes, dimensions and measures. Any concept can be used as a fact collection for its greater concepts and as a dimension for its lesser concepts. The notions of cube, dimensions and measure are supposed to be specific to concrete analysis scenarios and therefore are not part of the model.

The partially ordered schema can also be viewed as a generalization of star and snow-flake schemas where lesser concepts correspond to fact tables and greater concepts describe detail tables. The difference is that COM concepts do not have such predefined roles. A concept is a fact concept for its greater concepts and it is a detail concept for its lesser concept.

4.2. References for Modeling Containment

Traditional conceptual modeling distinguishes several relationships which describe how elements belong to each other or are composed of other elements: containment, aggregation/composition and part-of. They are used as independent semantic units which are not bound to the

VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.
http://www.cisjournal.org

internal structure of entities defined via attributes. Therefore, data modeling is broken into two branches: defining attribute structure of entities and defining relationships among entities.

COM eliminates this difference by using concept fields as elementary semantic units which also represent containment relation. It is achieved by providing a suitable interpretation of partial order relation rather than by introducing an additional independent mechanism. Partial order is interpreted in terms of containment by assuming that *lesser elements are contained (by-reference) in their greater elements*. Conversely, a greater element is a container for all its lesser elements. This interpretation means that references (modeled explicitly as identity-entity couple) are not simply a means of connectivity but rather are elementary semantic units: *to reference an element means to be included in it*. At the level of concepts, a greater concept is interpreted as a collection of its lesser concepts. For example, if a Book is characterized by a Publisher which is the type of one of its fields then this means that the Publisher is a collection of Books (Fig. 5). And if each Book is characterized by some publication Date then one Date element is a collection of Books (with this publication date).

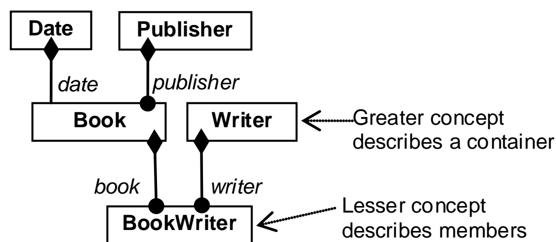

Figure 5. Partial order interpreted as containment

An important question is what are the differences between containment described by inclusion relation (Section 3) and containment described by means of concept fields. The main difference is that inclusion relation implements containment by-value where identities of child elements extend the parent identity. In this case, elements cannot change their parent because their identity is defined there. If a parent is deleted then all its children (extensions) are also deleted and in this sense it is semantically closer to composition. Therefore, inclusion should be used for defining permanent containment for identification purposes. Containment by-reference can be always changed by changing values stored in fields.

Partial order has also the opposite interpretation where an element is made up of its greater elements. For example, a Book is a combination of one Date *and* one Publisher. Thus an element is a combination of its greater elements (where it is a member) and a collection of its lesser elements (for which it is a set). The opposite character of these two relations is explicitly expressed via partial ordering but traditional models do not always distinguish between these two types of composition or introduce separate relations for them. It might be also useful to interpret partial order in logical and algebraic terms. In terms of logical connectives, an element is a conjunction of its greater elements and a disjunction of its lesser elements. In terms of algebraic operations, an element is a product of its greater elements and a sum of its lesser elements.

Partial order can also be interpreted via specialization-generalization relation by assuming that lesser concepts are more specific than their greater concepts. Note again that since partial order is implemented via references then this interpretation means that a referencing element is more specific than the referenced elements. It is quite natural because lesser concepts have some additional fields which make them more specific with respect to the greater concepts. By removing these additional fields we can reduce this concept to its greater concept so that the lesser concept IS-A greater concept. For example, the following concept is equivalent to its referenced (greater) concept because it does not define additional fields:

```
CONCEPT BookVariation // The same as Book
 IDENTITY
 ENTITY
  // Greater concept – more general
  Book book
```

If we add some new fields then it will be made more specific:

```
CONCEPT HeavyBook // More specific
 IDENTITY
 ENTITY
  // More general (greater) concept
  Book book
  DOUBLE weight
```

Generalization-specialization can also be modeled using inclusion relations as described in Section 3. The difference is that inclusion implements it by-value (by means of extension) and more specific elements cannot change their more general parent. In contrast, partial order is implemented by-reference and hence more general elements can be changed by changing properties of the more specific elements. Just as with other mechanisms, the choice of inclusion or partial order is a matter of (good) design.

4.3. References for Modeling Relationships

Thinking in terms of relationships is one of the most successful and wide spread data modeling design pattern implemented as a basic principle in many models including the Entity Relationship model (ERM) [5] and fact-oriented models [38]. Relationships are one of the key data modeling constructs intended as a means of defining associations among entities, that is, how entities are related to each other. However, one basic problem here is that it is not always easy to distinguish between entities and relationships because both may have properties, produce instances and even have relationships between relationships. In addition, relationships may well depend on the task being solved which makes it difficult to

VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.
http://www.cisjournal.org

maintain clear separation between a model and its applications as well as makes translation to a logical level ambiguous.

COM provides a solution to this problem by removing relationships as a separate data modeling construct. Instead, concepts can be interpreted as relationships with respect to other concepts. In other words, to be a relationship is a *relative* role of concepts in COM. This interpretation is defined in terms of partial order according to the following assumption: *lesser concepts play a role of relationships and dependencies between greater concepts*. A general data modeling pattern in this case is that if we have concepts and need to describe a relationship between them then it is necessary to introduce a new lesser concepts. For example (Fig. 6), if Book and Writer are known to be related (dependent) concepts then we simply add a new lesser concept BookWriter for describing this relationship. It is also easy to add a relationship between existing relationships because existing lesser concepts may have their own lesser concepts. For example, the Book concept in Fig. 6 relates two greater concepts Date and Publisher. However, Book is related with Writer via the BookWriter relationship. Note also that relationships are more specific than the concepts they relate.

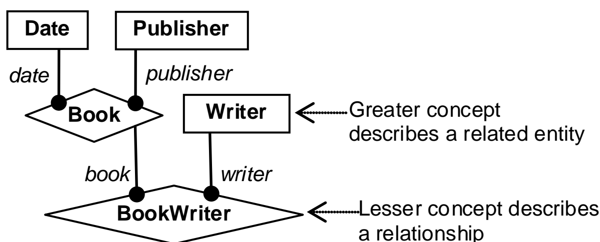

Figure 6. Partial order interpreted via relationships

The main benefit of this approach is that there is only one main data modeling construct, concept, but we are still able to represent two phenomena: entities and relationships. To be an entity or relationship is a role which is determined by the concept position in the partially ordered structure. It is analogous to the relative role of coordinates and points in the multidimensional interpretation (Section 4.1), and the relative roles of container and its elements in the containment interpretation (Section 4.2). Relationships represented by lesser concepts can be easily used for navigation [26] by means of a pair of de-projection and projection operations where we start from some concept, then go down to the relationship concept (possibly along several path segments) and finally go up to the target concept. For example (Fig. 6), if Books and Writers are connected via the relationship BookWriter (lesser concept) then all books of some writer can be retrieved as follows:

```
(Writer | name = 'Smith')
 <- (BookWriter) -> (Book)
```

Note that the BookWriter concept is between two arrows which is an indication that it is used as a (most specific) relationship. However, the Book concept can itself be treated as a relationship between its greater concepts Date and Publisher. Therefore it can be used to retrieve all publishers of the writer:

```
(Writer | name = 'Smith')
 <- (BookWriter) -> (Book) -> (Publisher)
```

Another advantage of this approach is that relationships may have different application-specific interpretations by retaining the original data structure unchanged. These application-specific interpretations can be defined via concept methods. For example, the BookWriter relationship can be used to get all authors of one book represented as a hasWriters relationship and implemented as follows:

```
CONCEPT Book
 IDENTITY
  CHAR(10) isbn
 ENTITY
  // Can be used as a field
  (Writer) hasWriters() {
   RETURN book <- (BookWriter)
          -> writer -> (Writer)
  }
```

The method hasWriters returns a collection of Writers which are found using the BookWriter concept interpreted as a relationship. First, this query uses de-projection operation for finding related BookWriter elements. Second, these elements are projected up to the Writer by finding all authors of this book. In other applications, we might need to define this relationship differently or to define additional relationships. This makes the model semantically more stable because it is less dependent on application-specific definitions of relationships.

5. DISCUSSION

Many existing data models are very similar in the way they represent real world entities. At least, most conventional models do not dispute that entities should be in the center of data modeling and by normally agreeing that they have to be uniquely identified and characterized by some properties. However, having only entities is not enough because some means of connectivity are apparently needed and this is precisely where most models differ by providing various mechanisms for structuring a set of entities:

- *Attributes* are characteristics of entities which are thought of as slots for containing values or references. If an attribute contains a reference then it can be used for connectivity. Almost all models provide this mechanism except for fact-oriented attribute-free models like the object-role model [13] and NIAM [38].
- *Relationship* is an independent data modeling construct which represents an association among several entities. This approach was first proposed in the entity-

VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.
http://www.cisjournal.org

relationship model [5] and then used in many other conceptual models.

- *Join* is a mechanism for finding related elements which contain the same values of attributes. This approach was first proposed in the relational model [6] but the idea of using common values for relating elements has its roots in logic-based methods. It is not widely used in conceptual modeling because join is a relatively low level operation used mostly at the level of queries.
- *Link* is an element of a binary relation which is an explicit representation of one connection between two elements. One formalization of this approach is provided in a family of knowledge representation languages called description logic. It is the basis of the Semantic Web and related standards and models like RDF.

In this context, COM can be characterized as an attribute-based model because concepts are defined in terms of attributes (called dimensions). However, attributes play much more important role in COM than in traditional models. First of all, COM does not distinguish between primitive, non-primitive and reference-valued attributes. There is only one kind of attributes which contain an identity representing some entity. Depending on what concept is used as the attribute type we can get different particular cases: a primitive or non-primitive value (entity class of the concept is empty), primitive or non-primitive reference (entity class of the concept is non-empty), primitive reference (identity class of the concept is empty). Note that references in COM are different from links because reference is a value which provides access to entity attributes while a link can be viewed as a special entity which itself has to be somehow represented. In other words, references are values which are passed by-copy while links (like RDF triples) are entities which contain data about the source element (subject), the property (predicate) and target element (object). COM attributes are similar to existential facts in object-role modeling. The difference is that they are modeled using the duality principle (internal structure of concepts) rather than as relationships. Importantly, COM does not treat references as a particular case of relationships. References in COM are a more basic notion which is embedded in the definition of concepts. Essentially, COM assumes that it is not possible to model real world things without modeling references just because reference definition is part of the thing definition. When defining a concept we simultaneously define some reference structure in its identity class (which can be empty in particular cases).

COM does not have a dedicated construct for representing domain-specific relationships and therefore it can be characterized as a relationship-free model. However, relationships still can be represented in COM by using concepts. The idea is that to be a relationship is a role of any concept with respect to its greater concepts. More specifically, a concept is regarded as a relationship for its greater concepts. Such a treatment of relationships allows us to avoid ambiguities when choosing between attributes and relationships as well as between entities and relationships. This approach is also more flexible because we avoid hard assignment of the role of relationship to the elements of the model so that relationships can participate in other relationships. COM principally distinguishes between two major notions and the corresponding mechanisms: *representing* things, and *relating* things. Things are represented by a value, called identity, the main function of which is providing access to the represented entity. Once represented, things can be related by using other things which store identities of the related elements. Note that an instance of a relationship is a normal element with its own identity which also can be related to other elements. On the other hand, a reference does not have its own separate identity and it is therefore not a binary relationship.

If relationships are intended for representing domain-specific associations between entities then semantic relations represent general purpose associations. There are several major semantic relations used in conceptual modeling and many their variations. The standard way to use semantic relations is to introduce a separate notation for each of them and it is actually the basis for contemporary conceptual modeling. An advantage is that such a model is independent of the implementation while a disadvantage is that its translation to a lower level can be quite ambiguous, requires high expertise, and eventually can be reduced to creating a completely new model. COM is different from conventional conceptual models in that it does not use a separate notation for semantic relations but rather provides an additional semantic interpretation to its basic constructs. The main advantage is that such a model is simpler because it can be used for both conceptual design and as a logical data model. However, this approach is more sensitive to the quality of the design because its basic constructs have a significant semantic load. For example, to assign a value to an attribute means to include this element to some set and to declare an attribute type means to specify an axis with coordinates for this concept instances.

One novel feature of COM is that it provides *two* versions for its main semantic relations. It is a consequence of having two major mechanisms for describing new elements:

- Extension is described by inclusion relation and allows for including new elements by-value
- Combination is described by concepts and allows for including a new element into several combined elements by-reference

For example, there are two ways to build a more specific element: either extending the base element (by including this concept into the super concept) or referencing the base element (by defining a new concept field with the type of the super concept). Semantically, to extend an element is equivalent to referencing it. The differences are in properties and uses of these two mechanisms. Extension is performed by-value which means that it is applied to identities (which are values) and

VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.
http://www.cisjournal.org

identities cannot change their parent. In contrast, combination is performed by-reference so that it is always possible to change the more general element (by setting the attribute value). It can be used to model multiple inheritance because each concept field actually defines a base element.

COM can be characterized as a relationship-free and attributed-based model as opposed to attribute-free models which rely on only relationships as a sole data modeling construct. Attribute-free models capture data semantics in terms of atomic (elementary or existential) fact types and represent them as relationships. Prime examples of the fact-oriented approaches are the Object-Role Model (ORM) [13], Natural language Information Analysis Method (NIAM) [38] and the Predicator Set Model (PSM) [14]. When compared to the fact-oriented models, COM makes a fundamental distinction between references and (binary) relationships. Although mathematically they are equivalent, they are treated differently from the data modeling point of view. More specifically, references are values which are able to provide access to other values (entity attributes). They do not have their own references and can be represented only directly by copying their contents. References always have a direction. In contrast, relationships do not have a direction and are not treated as a way to indirectly represent another entity. (In fact, a value can play two roles simultaneously: it can represent some entity and it can relate several elements stored in its attributes.) One difficulty with having only relationships is that we cannot say how they are represented. Indeed, if a relationship has three roles then how they are implemented? If these three roles are not implemented via (binary) relationships then what they are? In terms of diagrams, what are the lines representing roles if not binary relationships? Probably, the only answer is that we do assume the existence of some kind of primitive references but do not explicitly introduce them into the model because they are not supposed to be modeled. COM solves this difficulty by explicitly fixing a special status of references which are not relationships but rather provide a basic way to represent things and also can be viewed as roles. Yet, these references can be and should be modeled by describing complex domain-specific identities and roles. On the other hand, COM completely removes relationships from the model by using concepts instead of them. Therefore a diagram in COM has only two elements: lines representing references (or roles) and boxes representing concepts treated as either entities or relationships.

6. CONCLUSION

In this paper we discussed the principles of COM from the point of view of type modeling and conceptual modeling. These three principles are summarized below.

Duality. We argue that a data type should be broken into two equally important parts describing identities and entities. A model is then split into two orthogonal branches – identity modeling and entity modeling – by producing a nice yin-yang style of balance and symmetry between two sides of one reality. This allows us to make identities integral part of the model so that any entity must have an associated identity. A model can consist of only identities in the case it is intended for describing value types or an address space. Or it can describe only entities in the case it is aimed at modeling what is passed by-reference. In the general case, however, any type has both constituents and one can freely vary between by-value and by-reference semantics. The duality principle provides a novel view on the notion of reference because now a type is used to describe *both* parts: a reference and a referent. Another immediate benefit is a generalization and simplification of the type system the in relational and object-relational models because there is only one kind of domain without the necessity to distinguish between value domains and relations.

Inclusion. We also argue that it is enough to have one inclusion hierarchy to model hierarchical address spaces (where elements exist), containment (by-value) and inheritance (in a generalized form). This leads to an important principle: to include in a set means to inherit from this set and to be more specific than this set. Inclusion is also equivalent to identification with respect to the parent set which is implemented by extending the parent address. This approach also allows us to simplify data modeling because several mechanisms are described via one relation. In particular, it is possible to use inclusion as a conventional extension operator to add new properties and produce more specific data types. But this same relation is used to model hierarchical address spaces and containers. Another positive consequence of the inclusion principle is that we eliminate the asymmetry between classes (existing in hierarchy) and their instances (existing in flat space).

Partial order. And the third main point is that partial order relation can describe many existing mechanisms and semantic relationships:

- object-attribute-value - object is a lesser element and value is a greater element
- multidimensional space - point is a lesser element and coordinate is a greater element
- containment - greater elements are collections for lesser elements
- relationships - lesser elements relate greater elements

Data modeling is then reduced to partially ordering a set of concepts while other properties are derived from this structure. Importantly, the role of reference is revisited because they serve as elementary semantic units rather than a means of connectivity in a graph. In other words, the use of references is not limited by the possibility to retrieve elements and navigate through the graph structure. In COM, to reference an element means to specify a more general element, a coordinate for this element, a container for this element or a value for some attribute.

This work is a step towards developing a unified model which provides equal support for transactional, analytical

©2009-2012 CIS Journal. All rights reserved.

and conceptual views on data. COM takes a holistic view on data by unifying a wide range of existing data modeling approaches and reducing them to only three major principles. The main benefit of these principles is that it significantly simplifies data modeling by eliminating or reducing many incompatibilities which stem from a large number of diverse data modeling techniques and patterns.

REFERENCES

[1] M. Atkinson, P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys (CSUR)*, **19**(2), 105–70, 1987.

[2] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, North Holland, 1990.

[3] F. Bancilhon. Object databases. *ACM Computing Surveys (CSUR)*, **28**(1), 137–140, 1996.

[4] C. Chambers, D. Ungar, B. Chang, U. Hölzle. Parents are Shared Parts of Objects: Inheritance and Encapsulation in Self. *Lisp and Symbolic Computation*, **4**(3), 207–222, 1991.

[5] P.P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, **1**(1), 9–36, 1976.

[6] E. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, **13**(6), 377–387, 1970.

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

[8] S.S. Conn. OLTP and OLAP Data Integration: A Review of Feasible Implementation Methods and Architectures for Real Time Data Analysis. In *Proc. SoutheastCon*, 515–520, 2005.

[9] G.P. Copeland, D. Maier. Making Smalltalk a Database System. *ACM SIGMOD Record*, **14**(2), 316–325, 1984.

[10] K.R. Dittrich. Object-oriented database systems: the notions and the issues. In *Proc. Intl. Workshop on Object-Oriented Database Systems*, 2–4, 1986.

[11] F. Eliassen, R. Karlsen. Interoperability and object identity. *ACM SIGMOD Record*, **20**(4), 25–29, 1991.

[12] R. Finkelstein. MDD: Database reaches the next dimension. *Database Programming and Design*, 27–38, April 1995.

[13] T.A. Halpin. Object-Role Modeling: Principles and Benefits. *IJISMD*, **1**(1), 33–57, 2010.

[14] A. ter Hofstede, H. Proper, T. van der Weide. Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, **18**(7), 489–523, 1993.

[15] W. Kent. A Rigorous Model of Object References, Identity and Existence. *Journal of Object-Oriented Programming*, **4**(3), 28–38, 1991.

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

[17] W. Kim, J.F. Garza, N. Ballou, D. Woelk. Architecture of the ORION Next-Generation Database System. *TKDE*, **2**(1), 1990.

[18] R. Kimball, K. Strehlo. What's wrong with SQL. *Datamation*, June 1994.

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

[20] J. Pardillo, J.-N. Mazón, J. Trujillo. Bridging the semantic gap in OLAP models: platform independent queries. In *Proc. DOLAP '08*, 89–96, 2008.

[21] T.B. Pedersen, C.S. Jensen. Multidimensional database technology, *Computer*, **34**(12), 40–46, 2001.

[22] T.B. Pedersen. Multidimensional Modeling. *Encyclopedia of Database Systems*. L. Liu, M.T. Özsu (Eds.). Springer, NY., 1777–1784, 2009.

[23] D. Raymond. Partial order databases. *Ph.D. Thesis*, University of Waterloo, Canada, 1996.

[24] S. Rizzi, A. Abelló, J. Lechtenbörger, J. Trujillo. Research in data warehouse modeling and design: dead or alive? In *Proc. DOLAP'06*, 3–10, 2006.

[25] A. Savinov. Concept as a generalization of class and principles of the concept-oriented programming. *Computer Science Journal of Moldova*, **13**(3), 292–335, 2005.

[26] A. Savinov. Logical Navigation in the Concept-Oriented Data Model. *Journal of Conceptual Modeling*, Issue 36, 2005.

[27] A. Savinov. Grouping and Aggregation in the Concept-Oriented Data Model. In *Proc. ACM Symposium on Applied Computing (SAC'06)*, 482–486, 2006.

[28] A. Savinov. Query by Constraint Propagation in the Concept-Oriented Data Model. *Computer Science Journal of Moldova*, **14**(2), 219–238, 2006.

[29] A. Savinov. Concepts and Concept-Oriented Programming. *Journal of Object Technology*, **7**(3), 91–106, 2008.

[30] A. Savinov. Concept-Oriented Programming. *E-print: arXiv:0806.4746*, 2008.

[31] A. Savinov. Concept-Oriented Programming, In: *Encyclopedia of Information Science and Technology, 2nd Edition*, Editor: Mehdi Khosrow-Pour, 672–680, IGI Global, 2009.

[32] A. Savinov. Concept-Oriented Model. In V.E. Ferraggine, J.H. Doorn, & L.C. Rivero (Eds.), *Handbook*

VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.
http://www.cisjournal.org

of Research on Innovations in Database Technologies and Applications: Current and Future Trends, IGI Global, 171–180, 2009.

[33] A. Savinov. Concept-Oriented Query Language for Data Modeling and Analysis. In: *Advanced Database Query Systems: Techniques, Applications and Technologies*. L. Yan and Z. Ma (Eds.), 85–101, 2010.

[34] A. Savinov. Concept-Oriented Model: Extending Objects with Identity, Hierarchies and Semantics, *Computer Science Journal of Moldova*, **19**(3), 254–287, 2011.

[35] L.A. Stein. Delegation Is Inheritance. In *Proc. OOPSLA'87*, ACM SIGPLAN Notices, **22**(12), 138–146, 1987.

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

[37] M. Stonebraker. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, San Mateo, CA, 1995.

[38] G.M.A. Verheijen, J. van Bekkum. NIAM: An Information Analysis Method. *Information System Design Methodologies: A Comparative Review*. T.W. Olle, H.G. Sol and A.A.V. Stuart. North-Holland, Amsterdam, 537–590, 1982.

[39] R. Wieringa, W. de Jonge. Object Identifiers, Keys, and Surrogates - Object Identifiers Revisited. *Theory and Practice of Object Systems*, **1**(2), 101–114, 1995.