Computer Science Journal of Moldova, vol.**19**, no.3(57), 2011

Concept-Oriented Model: Extending Objects with Identity, Hierarchies and Semantics

Alexandr Savinov

Abstract

The concept-oriented data model (COM) is an emerging approach to data modeling which is based on three novel principles: duality, inclusion and order. These three structural principles provide a basis for modeling domain-specific identities, object hierarchies and data semantics. In this paper these core principles of COM are presented from the point of view of object data models (ODM). We describe the main data modeling construct, called concept, as well as two relations in which it participates: inclusion and partial order. Concepts generalize conventional classes by extending them with identity class. Inclusion relation generalizes inheritance by making objects elements of a hierarchy. We discuss what partial order is needed for and how it is used to solve typical data analysis tasks like logical navigation, multidimensional analysis and reasoning about data.

Keywords: Data modeling, object data models, set nesting, partial order, data semantics.

1 Introduction

The concept-oriented model (COM) is an emerging general-purpose approach to data modeling. It is aimed at unifying different views on data and solving a wide spectrum of problems in data modeling and analysis [31, 33]. COM overlaps with many existing data modeling methodologies but perhaps most of its features are shared with object data models (ODM) [10, 4, 3]. COM is not only based on the general principles of object-based models but can be viewed as their further development and generalization. If we take ODM as a starting point then what problems COM is going to solve? In other words, what are the main motivating factors behind this approach if it is considered a generalization of ODM? COM is aimed at solving the following three major problems:

How objects exist. COM provides a mechanism for *domain-specific references* so that *both* objects and their references may have arbitrary structure and behavior.

Where objects exist. COM turns each object into a set which is interpreted as a space, context, scope or domain for its member objects. The whole model is then turned into a *set-based* approach where sets are first-class elements of the model.

What objects mean. COM augments objects with *semantics* and makes references elementary semantic units. The meaning of an object is defined via other objects and this semantic information can be used for reasoning about data.

"Object identity is a pillar of object orientation" [16] and the role of identities has never been underestimated. There exist numerous studies [17, 38, 1, 16, 11] highlighting them as an essential part of database systems and arguing for the need in having strong and consistent notion of identity in data and programming models. A lot of identification methods have been proposed like primary keys [6], object identifiers [1, 17], l-values [18] or surrogates [12, 7]. Nevertheless, there is a very strong bias towards modeling entities while the support of identities is relatively weak. Identities have always been considered important but secondary elements remaining in the shadow of their more important counterpart. There is a very old and very strong belief that it is entity that should be in the focus of data modeling while identities simply serve entities. Almost all existing data (and programming) models assume that identities should be provided by the platform and there is no need for modeling domain-specific identities. For this reason, the roles of entities and identities are principally separated: entities have *domain-specific* structure and behavior while identities have *platform-specific* structure and behavior.

COM changes this traditional and currently dominating view by assuming that identities and entities are equally important for data modeling. In this context, the main goal of COM is to provide data modeling means where *both* identities and entities may have arbitrary domain-specific structure and behavior. To solve this problem, COM introduces a novel data modeling construct, called *concept* (hence the name of the model), which generalizes classes. Its main advantage is that it allows for modeling arbitrary domain-specific identities (references) what is impossible if conventional classes are used. Importantly, identities and entities are modeled together as two parts of one thing. Elements in COM can be compared to complex numbers in mathematics which also have two constituents (real and imaginary parts) manipulated as one whole.

Set-orientation is one of the most important features of any data model just because data is normally represented and manipulated as groups rather than as individual instances. Probably, it is the solid support of set-oriented operations why the relational model [6] has been dominating among other data models for several decades. And insufficient support of the set-oriented view is why object-oriented paradigm is less popular in data modeling than in programming. Of course, we can model sets, groups or collections manually (at conceptual level) but in this case the database management system is unaware of these constructs and cannot help in maintaining consistency of these structures. In this context, the main goal of COM is to turn instance-based view into a set-based model where set is a first-class notion supported by the model at the very basic level. The idea here is that any instance has to be inherently a set, and vice versa, a set has to be a normal instance. To solve this problem, COM introduces a novel relation, called *inclusion*. The most important change is that elements exist within a hierarchy where any parent is a *set* of its children and any instance is a member within its parent superset. Parent elements are shared parts of children and are treated as a space, context, scope or domain for its children. In contrast, in most class-based models classes exist within a hierarchy while their instances exist in flat space (so we do not have a hierarchy of instances). The use of inclusion relation makes COM similar to the hierarchical model [37] where parents are containers for their child elements. In addition, inclusion generalizes inheritance and can be used for reuse, type modeling and other purposes. This approach is also very close to prototype-based languages [5, 19, 34] where parents are shared parts of children. However, these languages do not use classes while COM allows for using both classes (in the form of concepts) and object hierarchies (in the form of inclusion).

According to Stefano Ceri [20], "the three most important problems in Databases used to be Performance, Performance and Performance; in the future, the three most important problems will be Semantics, Semantics and Semantics". The main advantage of having semantics in databases is that it "should enable it to respond to queries and other transactions in a more intelligent manner" [7] by providing richer mechanisms and constructs for structuring data and representing complex application-specific concepts and relationships. There has been a tremendous interest in semantic models [13, 21] but most of the works propose conceptual models which need to be translated into some logical model. The lack of semantics in logical data models significantly decreases their overall value and, particularly, decreases the possibility of information exchange, data integration, consistency and interoperability.

As a response to this demand, COM proposes a novel approach to representing and manipulating semantics as integral part of the model. The main idea is that concepts are partially ordered by using the rule that a field type represents a greater concept. As a result, all instances exist as elements of a *partially ordered set* where they have greater elements and lesser elements. In contrast, most other models are graph-based with objects treated as nodes and references considered the edges. In COM, references always represent greater elements by playing a role of an elementary semantic construct rather than a navigational tool in a graph. Partial order changes the way how data is being accessed: instead of using graph navigation, COM introduces two operations of *projection* and *de-projection*. Although there exist approaches which are based on partial order relation [24], only in COM it is used as a primary relation for representing data semantics and reasoning about data. The main benefit of using partial order is that it "seems to fulfill a basic requirement of a general-purpose data model: wide applicability" [24], that is, many conventional data modeling mechanisms and patterns can be explained in terms of this formal setting. In COM, this approach was shown to be successful in solving such highly general tasks as logical navigation [27], multi-dimensional analysis [26], grouping and aggregation [28], constraint propagation and inference [29].

In the next three sections we discuss three principles of COM. In Section 2 we describe duality principle and introduce the main data modeling construct, concept. In Section 3 we discuss inclusion principle by defining inclusion relation among concepts and showing its differences from inheritance. Section 4 discusses order principle by demonstrating how partial order can be used for data access and solving typical data modeling tasks. Section 5 makes concluding remarks. We will follow a convention that identities are shown as gray rounded rectangles while entities are white rectangles. Also, concepts names will be written in singular while collection names will be in plural.

2 Concepts Instead of Classes

Existence precedes and rules essence — Jean-Paul Sartre, Being and Nothingness

2.1 Modeling Identities

How things exist and what does it mean for a thing to exist? In many theoretical and practical contexts existence can be associated with the notions of *representation* and *access*. This means that a thing is assumed to exist if it can be represented and accessed. Conversely, if a thing does not have a representation or cannot be accessed then it is assumed to be non-existing. According to this view, any existing thing is supposed to have a unique *identity* which manifests the fact of its existence.

One of the main distinguishing features of COM is that it makes identities first-class elements of the model. To describe how COM differs from other identification schemes we will use the following three dimensions:

- Strong vs. weak identities
- Domain-specific vs. platform-specific identities
- Implicit vs. explicit identities

The first criterion divides all identification schemes in two groups [38, 11]: *strong identities* and *weak identities*. Strong identities exist separately from the identified entity while weak identities are actually internal properties of the entity used for identification purposes (identifier properties).

Memory address is an example of a strong identity because memory cells do not contain their address as a property (otherwise memory would be a two-column array with addresses in the first columns and cells in the second column). Object references are also strong identities because they are not contained in object fields. In particular, if a class does not have fields then its instances have zero size but still have references. In contrast, primary keys in the relational model (RM) are weak identities because they are made up of a number of normal attributes.

The main problem with weak identities like primary keys is that it is not possible to access entity properties without some kind of strong identity which therefore should be provided by a good data model. In other words, some kind of strong identity must always exist while having weak identities in the model is optional. In this sense, weak identities should be viewed as a design pattern rather than a primary data modeling mechanism.

Strong identities are provided in ODM and COM because objects in these models are represented by a separable reference (Fig. 1). In contrast, RM relies on weak identities in the form of primary keys. Note however that many implementations provide some kind of strong identity for representing rows like surrogates and their usefulness is theoretically justified.

The second dimension for evaluating various identification schemes

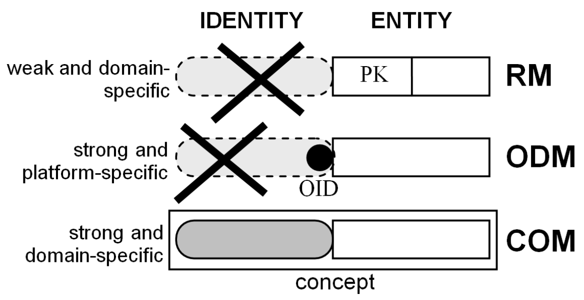


Figure 1. Modeling identities and entities in different models

is the possibility to model *domain-specific* (user-defined) identities as opposed to having only *platform-specific* (primitive, system) identities. An example of platform-specific identity is OID, surrogate or object reference, which are provided by the DBMS. The mechanism of primary keys is an example of domain-specific identities which is defined by the data modeler and reflects specific features of the problem domain. Since identities are integral part of the problem domain, a good data model should provide an adequate mechanism for their modeling. In particular, this mechanism is available in RM and COM but not in ODM (Fig. 1).

The third criterion separates all identification mechanisms depending on whether they provide *implicit* or *explicit* identities. An identity is called explicit if its functionality has to be used manually to access the represented entity. For example, primary keys are explicit identities because it is necessary to manually specify join conditions. The problem of explicit identities is that one and the same fragments span many queries. If the structure of the identity changes then all queries involving this fragment have to be also updated. For example, all queries where we need to get bank accounts given a set of persons involve the same fragment with the join condition like the following:

```
WHERE ACCOUNTS.owner = PERSONS.id AND ...
```

Note that this fragment is mixed with other conditions. What is worse, if we change the primary key then all queries involving it in their join conditions have to be also updated (join is a cross-cutting concern).

In contrast, implicit identities hide their structure and functionality. For example, each time an object is about to be accessed, the DBMS needs to resolve its OID, then to lock memory handle and finally execute the operation using the obtained memory address. If it is a remote reference then the access procedure is even more complex. However, all these operations in queries are hidden – it is enough only to specify a variable and operation to be executed – all the intermediate actions will be executed behind the scenes. The mechanism of implicit identities has numerous advantages and therefore it should be supported by good data models. However, dot notation is also rather restrictive in comparison with the flexibility of joins. In COM and ODM, implicit identities are supported at basic level while RM relies on explicit identities in the form of join conditions for FK-PK pairs.

2.2 Identity and Entity — Two Sides of One Thing

To provide strong domain-specific implicit identities COM assumes in its duality principle that *any element is a couple consisting of one identity and one entity.* Formally, an element is represented as a couple of one identity tuple and one entity tuple. To model such identity-entity couples COM introduces a novel construct, called *concept.* Concept is defined as a couple of two classes: one identity class and one entity class. Both classes may have fields which are referred to as *dimensions* (to emphasize their special role for defining partial order described in Section 4) as well as other members like methods and queries. Importantly, these two constituents cannot be used separately because identity-entity couples are regarded as one whole. When a concept is instantiated we get a couple consisting of one identity and one entity so that the whole approach is reduced to manipulating identity-entity couples. Identity is an observable part manipulated directly in its original form. It is passed by-value (by-copy) and does not have its own separate representative. Entity can be viewed as a thing-in-itself or reality which is radically unknowable and not observable in its original form. The only way to do something with an entity consists in using its identity as an intermediate. Identities are transient *values* while entities are persistent *objects*.

For example, a bank account in COM can be described by the following concept:

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

Identity class of this concept has one dimension which stores account number and entity class has one dimension which stores the current balance.

Concepts generalize conventional classes and are used instead of them in COM. There are two special cases:

- identity class is empty
- entity class empty

If identity class is empty then such concept is equivalent to a conventional class. Its instances will be represented by identities inherited from the parent concept (see Section 3). For example, we could define a class of color objects as follows:

```
CONCEPT ColorObject
   IDENTITY // Empty
   ENTITY
      INT red, green, blue
```

Instances of this concept are normal objects represented by some kind of platform-specific reference or OID. If all concepts have empty identity classes then we get the object-oriented case where one and the same platform-specific identity is inherited from the root concept and is used to represent *all* entities. What is new in COM is that it allows for modeling user-defined types of references together with the represented entities. Such references can be thought of as user-defined surrogates or separable primary keys. If entity class is empty then this concept describes a value type (value domain). Indeed, its identity does not have entity part and hence it is not intended to represent anything (if it is not used as identity in a child concept). For example, we could define colors as values:

```
CONCEPT ColorValue
   IDENTITY
      INT red, green, blue
   ENTITY // Empty
```

Any instance of this concept is a value which is copied when it is passed or stored.

Concepts are instantiated precisely as classes by specifying their name as a type of the variable and then invoking this concept constructor. Moreover, from the source code where concepts are used, we cannot determine if it is a concept-oriented query or an object-oriented one – it depends how the concepts are defined. If they have only one constituent (entity) then it is an object-based approach and if they have two constituents (identity and entity) then it is a concept-oriented approach.

In data modeling concepts are normally used to declare types of elements in collections where data are stored. For example, if data are stored in tables then concepts are used to parameterize such a table by specifying the type of its elements. In an SQL-like language a table for storing bank accounts could be created as follows:

```
CREATE TABLE Accounts CONCEPT Account
```

Here `Accounts` is a table name and `Account` is a concept name so that this new table will store only elements of this concept type. Note that collections in COM are different from relational tables because they have domain-specific row identifiers. Such a collection can be viewed as a memory with arbitrary user-defined addresses and arbitrary user-defined cells.

2.3 Concepts for Domain Modeling

COM provides means for describing both values (identity class) and objects (entity class). The specific feature is that they are described and then exist in couples. In contrast to object-relational models (ORM) [35, 36] where value domains and relations are modeled separately, COM provides one unified construct for modeling simultaneously value and relation types. In other words, there is only one type – a type of an element – but this element has a transient part (value) and a persistent part (object). The main benefit is that we can vary the "degree of persistence" by choosing which attributes belong to transient part (identity) and persistent part (entity).

In ORM, the idea is that relation attributes can store complex values rather than only primitive values. For example, we can define a user defined type composed of two integers:

```
TYPE MyType
   field1 AS INT
   field2 AS DOUBLE
END TYPE
```

After that it can be used as a type of relation attributes:

```
RELATION MyRelation
   intAttribute AS INT
   myAttribute AS MyType
END RELATION
```

The problem is that traditional approaches to data and type modeling do not allow us to use relations as types:

```
TYPE NewType
   ~~myField AS MyRelation~~ // Not possible
END TYPE
RELATION NewRelation
   ~~myAttribute AS MyRelation~~ // Not possible
END RELATION
```

In other words, relations are defined on domains only (primitive or complex) but not on other relations.

COM solves this problem by using concepts which essentially unite domain modeling and relation modeling. In COM, there is only one kind of domain or set – it is a set of concept instances which are value-object couples. Once a concept has been defined, it can be then used as a type in any other concept independent of its use for value modeling, relation modeling or mixed use. For example, we could define colors as values:

```
CONCEPT Color
   IDENTITY
      INT red, green, blue
   ENTITY // Empty
```

This concept is equivalent to a user-defined type. The difference is that new fields can be added to either value part (identity class) or object part (entity class). For example, if each color has a unique name which has to be shared then we add the corresponding field to the entity class:

```
CONCEPT Color
   IDENTITY
      INT red, green, blue
   ENTITY
      CHAR(64) name
```

It is already a mix of three primitive values in the identity class and one object field. Now suppose that colors may have some other property which is supposed to be stored in a separate relation:

```
CONCEPT ColorDescr
   IDENTITY
      INT code
   ENTITY
      CHAR(2) lang
      CHAR(64) name
```

Now the color concept simply declares a field with this concept as a type:

```
CONCEPT Color
   IDENTITY
      INT red, green, blue
   ENTITY
      ColorDescr descr
```

Importantly, only one concept name is used as a type of variables in the model independent of its division on identity and entity parts.

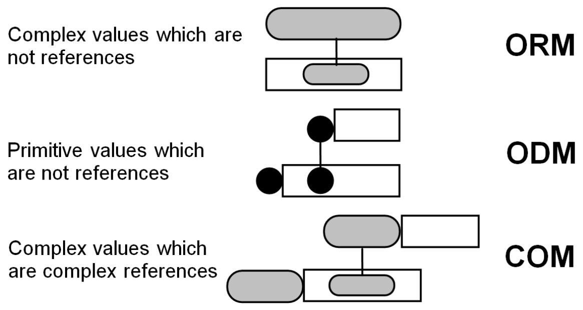

Complex values which are not references
ORM
Primitive values which are not references
ODM
Complex values which are complex references
COM

Figure 2. Value vs. object types in different approaches

All domains (independent of whether they are values, objects or both) are described using one system of types. If we use only identity classes then all elements in the model are values and it can be used for value modeling precisely as it is done in ORM. If all concepts have only entity part then we can model objects represented by OIDs or relations where rows are represented by surrogates. But in the general case, and it is one of the main advantages of COM, we can freely vary the division between values and objects. The differences between ORM, ODM and COM are shown in Fig. 2. In ORM, we can model only value domains which are used as attribute types. In ODM, we can model only object types. In COM, we can model both value types and object types.

3 Inclusion Instead of Inheritance

A place for everything and everything in its place — Victorian proverb

3.1 Modeling Hierarchies

In the previous section we asked the question *how* entities exist and assumed that the fact of their existence is manifested by means of identities. In this section, the main question is *where* objects exist. COM assumes that if something exists then there has to be some space, environment, domain, container or context to which it belongs, that is, things are not able to exist outside of any space. Assuming so, the next question is what does it mean to exist *within* some space? The answer is that existence within a space means that the element is identified with respect to this space. Further assuming that the space is a normal element and any element is included only in one space then we get inclusion principle: *elements exist within a hierarchy where any element (excepting the root) is included in one parent with respect to which it is identified.*

Hierarchies are used in almost all branches of science and their descriptive role can hardly be overestimated. They exist almost everywhere in real life and they are especially useful in programming and data modeling. However, there exist many interpretations of hierarchies in terms of different relations:

- Containment hierarchies (inclusion relation)
- Re-use hierarchies (inheritance relation)
- Semantic hierarchies (specialization-generalization relation)

Containment hierarchies are used in the hierarchical data model [37] where many child elements are *included* in one parent and exist in its context. Hierarchies are also one of the corner stones of the object-oriented paradigm where they exist in the form of inheritance relation. Here the idea is that one base class can be extended by many more specific classes and in this way we can describe arbitrary entities from the problem domain by *reusing* more general descriptions. In semantic and conceptual models hierarchies are used to describe abstraction levels of the problem domain by defining more specific elements in terms of more general elements.

Yet, in spite of the highly general and natural character of the hierarchical view on data, it has not got a dominant position in data modeling. One reason for this state of affairs is asymmetry between the space of classes and the space of their instances [34]: classes exist within a hierarchy while their instances (objects) exist in flat space. Classical inheritance considers only class hierarchies and it is not possible to produce a hierarchy of their instances precisely what is needed in data

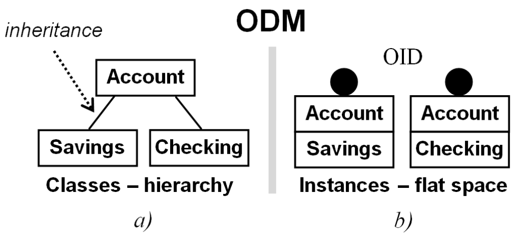


Figure 3. Asymmetry between classes and instances

modeling (and implemented in the hierarchical model). In other words, by using inheritance we cannot store objects in hierarchies like in the hierarchical model and it is a strong limitation because a database is modeled as a large *flat* space of objects.

For example, if two classes `Savings` and `Checking` extend one base class `Account` then we get a hierarchy where the parent class is shared among its child classes (Fig. 3a). Surprisingly, instances of these classes exist in a flat space so that each extension has its own parent and each instance is identified by one kind of OID (Fig. 3b). In other words, if two child classes have one shared parent class then their instances have no shared parent.

The conception of data reuse exists only at the level of classes but not their instances (code is reused in both cases). For data modeling purposes, it would be natural to create one main account object and then several savings and checking accounts within it. However, it is not possible using the traditional view on hierarchies, inheritance and semantic relationships. The goal of COM here is to unite the existing interpretations of hierarchies by introducing one relation for modeling containment (like in the hierarchical data model), inheritance (like in object data models), and generalization-specialization relation (like in semantic data models).

3.2 Inclusion for Modeling Hierarchies and Inheritance

COM revisits hierarchies by introducing a new relation, called *inclusion.* Inclusion relation assumes that child concepts are declared to be included in the parent concept. The most important property is that concept instances also exist within a hierarchy so that parents are shared parts of children. Note that the possibility to have many children within one parent is implied by the mechanism of identities. Each child is an instance of its concept and hence has some identity. This identity is always relative to the parent instance which has its own identity. For example, assume that concept `Savings` is included in concept `Account` (Fig. 4a):

```
CONCEPT Savings IN Account
   IDENTITY
      CHAR(2) subAccNo
   ENTITY
      DOUBLE balance
```

This concept declares its identity class having one dimension storing savings account number and its entity class having one dimension storing the current balance of the sub-account. It is important that sub-account number is a *relative* number which is meaningful only in the context of its main account. Hence (Fig. 4b) many savings accounts (instances of concept `Savings`) can be created in the context of one main account (an instance of concept `Account`). In particular, main account fields are shared among many savings accounts. In contrast, if `Savings` were a class inheriting from class `Account` then any new instance of `Savings` would get its own main account with all its fields.

An interesting observation about COM is that with the introduction of inclusion it essentially revives the hierarchical model of data but at the same time remains compatible with the object-oriented view. Inclusion hierarchy can be viewed as a nested container where each object has its own hierarchical domain-specific address which is analogous to a normal postal address. For example, a savings account might be identified by a reference consisting of two segments: $\langle '123456789' \rangle : \langle '02' \rangle$ where the first number is the main account and the second number

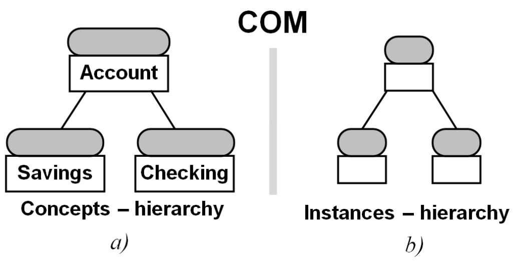


Figure 4. Symmetry between concepts and instances

identifies the relative savings account.

Note that these object identities are *virtual addresses* because they are defined in domain-specific terms which are not directly bound to the platform. Thus the represented entities can reside anywhere in the world. In particular, bank account objects can be persistently stored in a database and the account number is used to retrieve them. It is possible to specify how these virtual addresses are translated to physical addresses [30]. This can be used for implementing such mechanisms as replication, partitioning and distribution.

An important use of COM inclusion hierarchies consists in modeling types of values. If we define a concept without entity class then it will describe some value type. Using inclusion relation, this value can be extended by adding new dimensions and producing a more specific type. In this way we can build a hierarchy of value types. For example, we could define colors as elements identified by their unique name and having three entity properties:

```
CONCEPT Color IN ColorObject
   IDENTITY
      CHAR(16) name // For example 'red'
   ENTITY // Inherited from ColorObject
```

Of course, we could use conventional classes for this purpose like it is done in ORM but then we would need to have two kinds of classes for values and objects. The distinguishing feature of COM is that concepts are intended for describing *both* values and objects as couples. In relational terms, COM allows for modeling *two* type hierarchies for domains and relations using only one construct (concept) and one relation (inclusion). In the general case, a domain in COM is a set of identity-entity couples. In this sense, it is a step in the direction of unifying object-oriented and relational models by uniting two orthogonal branches: domain modeling and relational modeling.

If concepts have empty identity classes then inclusion is a means for modeling relation types. For example, an existing concept `Persons` can be extended by introducing a new concept `Employees`:

```
CONCEPT Employees IN Persons
  IDENTITY // Empty
  ENTITY
    DOUBLE age
```

Due to the nature of inclusion relation, many employee records can belong to one person record which is obviously not what we need in this situation. In order to model classical extension, identity class of the `Employee` concept is left empty. In this case no more than one employee segment is possible because of the absent identity class. As a result, each person record will have one or zero extensions which means that a person belongs to either base `Person` concept or to the extended `Employee` concept.

4 Partial Order Instead of Graph

Ordnung muss sein — German phrase

4.1 Partial Order for Data Modeling

In the previous sections we described how objects exist and where they exist. In this section we answer the question what an object *means*, that is, what is its semantic content. When an element is created it gets some identity which determines its location in the hierarchical address space but says little about its relationships with other elements. To semantically characterize an element, it has to be connected with other elements in the model. Almost all data models including ODM are implicitly or explicitly graph-based, i.e., they assume that a set of data elements is a graph. In this case semantics is encoded in relationships among elements. The distinguishing feature of COM is that it uses an approach where connectivity and semantics are represented by means of *partial order relation*, i.e., a database is defined as a partially ordered set of elements. As a result, any element participates in *two* orthogonal relations simultaneously: inclusion and partial order.

Strict partial order is a binary relation $<$ (less then) defined on elements of the set and satisfying the properties of irreflexivity and transitivity. If $a < b$ (a is less than b) then a is a lesser element and b is a greater element. In diagrams greater elements are positioned higher than lesser elements. Given two data elements, the main question in COM is whether one of them is less than the other. If we do this comparison for all elements then we get a concept-oriented database. Thus a concept-oriented database is a partially ordered set an example of which is shown in Fig. 5 where b, d and f are the greatest elements, and a and c are the least elements.

Formally, partial order in COM is represented by means of tuples by assuming that a tuple is less than any of its members: if $a = \langle \dots, e, \dots \rangle$ then $a < e$. For example, element a in Fig. 5 is less than b, d and e just because it is defined as $a = \langle b, d, e \rangle$. Thus if we have a number of tuples defined via each other then they induce partial order, and vice versa, if there is partial order then it can be represented by tuples representing elements and their members representing greater elements.

In object-oriented terms, this principle means that *references represent greater objects*. If one object references another object via one of its fields then the first object is less than the referenced object. For example, if a book element is an object storing a publisher, title and sales in its fields then these three constituents are its greater elements. Semantically, they are considered more general terms which define the meaning of the more specific book element. If we change a greater element then the meaning of all lesser elements which use it will also change.

To describe a partially ordered structure of elements at the level of

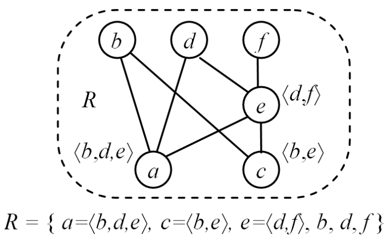


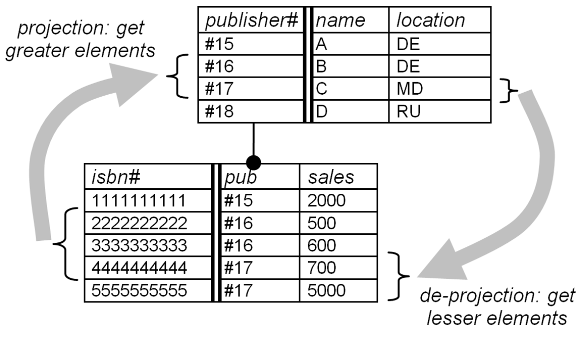
Figure 5. Concept-oriented database is a partially ordered set

concepts, COM uses the following principle: *dimension types represent greater concepts*. A set of concepts (with inclusion relation) partially ordered using this principle is referred to as a *concept-oriented schema* and a set of their instances is referred to as a *concept-oriented database*. Note that concepts and their instances participate simultaneously in *two* relations: inclusion and partial order.

For example, assume that concept `Book` has a dimension referencing its publisher of type `Publisher`:

```
CONCEPT Book // Book < Publisher
   IDENTITY
      CHAR(10) isbn
   ENTITY
      Publisher pub // Greater concept
      DOUBLE sales
```

According to this definition, `Book` is a lesser concept and `Publisher` is a greater concept: `Book < Publisher`. In object oriented approach, field types constrain possible elements that can be referenced via this field. In COM, we not only constrain possible referenced elements but also say that the referenced element is greater than this one and this property is then used for querying and reasoning about data.


projection: get greater elements
de-projection: get lesser elements

publisher#	name	location
#15	A	DE
#16	B	DE
#17	C	MD
#18	D	RU

isbn#	pub	sales
1111111111	#15	2000
2222222222	#16	500
3333333333	#16	600
4444444444	#17	700
5555555555	#17	5000

Figure 6. Projection and de-projection operations

4.2 Projection and De-Projection Operations

Data access operations in COM rely on the partially ordered structure of concepts and their instances. If we move up in the direction of some greater concept then this operation is called *projection* and denoted by right arrow `'->'`. If we move down in the direction of a lesser concept then this operation is called *de-projection* and denoted by left arrow `'<-'`. Given a set of elements, projection returns all greater elements and de-projection returns all lesser elements along the specified dimension. A sequence of projections and de-projections is called a *logical access path*. The main difference from graph-based models is that these operations are intended for changing the level of detail by moving only up and down in a zig-zag manner along the structure of dimensions. This makes COM similar to multidimensional models [22].

For example, assume that any book (`Books` collection) has one publisher (`Publishers` collection). Publishers of all books with low sales can be found by projecting the selected `Books` to the greater `Publishers` collection (Fig. 6):

```
(Books | sales < 1000) -> pub -> (Publishers)
```

All books of the selected publishers can be found using de-projection:

```
(Publishers | name == 'C') <- pub <- (Books)
```

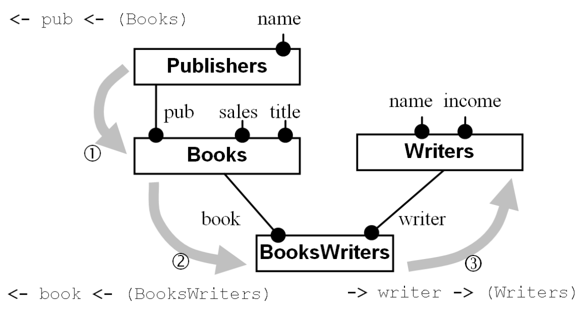


Figure 7. Logical navigation and inference in COM

Note that projection is a *set* of elements from the target domain without repetitions. For example, we could get all locations of the publishers as follows:

```
(Publishers) -> location
```

Here we do not write the primitive target domain because it can be restored from the schema and therefore is optional. The result of this query for the example in Fig. 6 consists of three values 'DE', 'MD' and 'RU' even though the value 'DE' occurs two times.

This approach is very convenient for retrieving related elements in complex schemas. Let us consider a schema in Fig. 7 with many-to-many relationship between books and writers where each book belongs to one publisher. All writers of the selected publisher can be retrieved for three steps using the following query:

```
(Publishers | name = 'XYZ')
   <- pub <- (Books)                           (1)
   <- book <- (BooksWriters)                   (2)
   -> writer -> (Writers)                      (3)
```

Here the selected publisher is (1) de-projected down to the collection of books, then (2) further down to the `BooksWriters` collection, and (3) finally the constrained facts are projected up to the `Writers` collection.

It is very easy to write correlated queries using this approach. For example, assume that we need to find all books with sales higher than the average sales in their respective publishers. The average sales figure for a publisher is computed as follows:

```
AVG( pub <- (Books).sales) )
```

Now we simply select books which have sales higher than this number:

```
(Books | sales > AVG(pub <- (Books).sales) )
```

For comparison, in SQL this query has a rather complicated and not very intuitive form:

```
SELECT isbn FROM Books b WHERE
   sales > (SELECT AVG(sales) FROM Books WHERE
      pub = b.pub )
```

One advantage of our approach is that it does not use joins and therefore queries in COM are much simpler than in SQL. In COM, it is possible to use the conventional dotted notation however it does not remove the necessity to use joins. Therefore, dotted notation is used only when it is necessary to access individual elements. For set-oriented operations COM relies on projection and de-projection. An advantage is that these operations hide the underlying structure of identities and it is not necessary to change all queries if some identity has changed.

4.3 Reasoning about Data

An important consequence of having partial order with projection and de-projection operations is the ability to reason about data by *automatically* deriving conclusions from initial constraints. It is not necessary to specify an exact access path because the system is able to propagate initial constraints automatically. In traditional semantic and deductive models this can be done using inference rules which are treated differently and managed separately from data. In COM, data itself is treated inherently as dependencies and data schema is used for constraint propagation. Inference procedure is essentially integral part of the model because it does not rely on inference rules but rather uses data for reasoning about data. In other words, once a model has been defined, it can be immediately used for inference.

In our example it is possible to derive writers belonging to the selected publisher without specifying *how* it has to be done. Such a query provides only (i) what we have in the form of source constraints and (ii) what we want to get by specifying a target collection. The source constraints are then propagated to the target automatically by the system. For example, such a query could be written as follows:

```
GIVEN (Publisher | name = 'XYZ')
GET (Writers)
```

The most important feature of this query is that it does not specify *how* the result has to be obtained and therefore this query can be answered by only using some semantic data model and its ability to infer the result. Most semantic models are based on formal logic where the result is derived using inference rules. COM proposes an alternative approach to inference which relies on partially ordered structure of the database [29]. This procedure consists of the following two steps:

Down. Source constraints X are propagated down to the bottom collection Z using de-projection: $X \leftarrow\star Z$

Up. The constrained bottom collection Z is propagated up to the target collection Y using projection: $Z \star\rightarrow Y$

Here operators '$\leftarrow\star$' and '$\star\rightarrow$' (with star symbol) denote de-projection and projection along *all* dimension paths. Using this two-step inference procedure we can get all authors of one publisher using the following semantic query:

```
(Publishers | name = 'XYZ')
  <-★ (BooksWriters) ★-> (Writers)
```

Note that this query does not involve any dimension name. It says only that the inference has to be carried out using `BooksWriters` as a bottom collection. Thus the chosen `Publishers` are de-projected down to `BooksWriters` and then up to `Writers` which are returned as a result collection. Yet, this query can be further simplified if we unite two steps into one inference operator denoted `'<-★->'`. This operator works with only source constraints and a target:

```
(Publishers | name = 'XYZ') <-★-> (Writers)
```

To illustrate how rather complex queries can be written in a very concise form let us assume that we want to consider only small publishers (with less than 10 books) and return only writers who have written more than 2 books:

```
(Publishers p | p <- pub <- (Books) < 10)
   <-*-> (Writers w | w <-*-> (Books) > 2)
```

Here we use a shortcut that comparison of a collection with a number implies COUNT aggregation function, that is, the condition `p<-pub<-(Books) < 10` is true if the publisher has less than 10 books, and the condition `w <-*-> (Books) > 2` is true if the writer has more than 2 books.

4.4 Multiple Propagation Paths

Generally, there are two approaches to constraint propagation:

Direct. Source constraints are imposed directly on the elements of the source set and then they are propagated through the model.

Inverse. Source constraints are expressed in terms of the target elements and then imposed on them so that target elements are selected by specifying properties they have to satisfy.

Projection and de-projection operations are an example of the direct approach while SQL is an example of the second approach. One limitation of the constraint propagation procedure via projection and de-projection operations is that we cannot use many source constraints. For example, assume that the task is to find all books belonging to the selected publishers and writers (Fig. 7). In this case we have two constraints: a set of publishers and a set of writers. One solution is to use the second (traditional) approach by simply expressing these two source constraints as properties of the books:

```
(Books p | p.pub.name = 'Springer' AND
   p <-*-> (Writers | name = 'Smith') > 0 )
```

This query involves two explicit conditions imposed directly on the retrieved elements. The first condition selects books depending on their publisher and the second condition selects books depending on their author. Then these two conditions are combined using logical 'AND' operation.

To solve this problem using the direct approach source constraints can be independently propagated and then the result is built as their intersection (also denoted by 'AND'):

```
(Publishers | name='Springer') <- (Books) AND
(Writers | name = 'Smith') <-*-> (Books)
```

This query consists of two propagation paths leading to the same target collection. The first operation propagates publishers to books and the second operation propagates writers to books. Then these two sets are combined using set intersection operation.

Strictly speaking the above query also contains a portion with inverse constraints where the names of publishers and writers are specified. These fragments can be removed by rewriting this query as follows:

```
'Springer'<- name <- (Publishers)
  <- pub <- (Books) AND
'Smith'<- name <- (Writers)
  <- writer <- (BooksWriters) -> book -> (Books)
```

The most important property of this query is that it does not involve inverse constraints at all. Both of its access paths start from some value and lead to the same target collection. The values are taken from primitive domains (text strings in this example). In contrast, inverse queries specify constraints as properties of collection elements. Of course, such a query is too verbose and in practice these two approaches are combined. For us it is important to understand that COM supports both approaches.

4.5 Analytical Queries

In previous sections we described how data can be retrieved from the database. In analytical applications, it is necessary to have a possibility to compute new values using existing data. For this purpose, COQL introduces CUBE operator (also denoted as FOREACH in other papers) which combines several source collections and returns their product:

```
CUBE(Collection1, Collection2, ...)
BODY {
   ...
}
RETURN ...
```

This query allows us to iterate over all combinations of elements in the source collections and to perform intermediate calculations in its BODY block. The structure of the result is specified in the RETURN statement.

For example (Fig. 8), assume that each book belongs to some genre and writers live in certain countries. The goal is to show how book sales are distributed among genres and countries. The difficulty is that a book may have many authors living in different countries and we want to distribute the book sales evenly among the authors. To solve this problem, a derived method is added to the BooksWriters concept which computes sales for one book and one author:

```
CONCEPT BooksWriters
   IDENTITY ...
   ENTITY
      DOUBLE sales() {
         RETURN book.sales / book <- (BooksWriters)
      }
```

Here sales can be considered a normal (read-only) dimension which is computed for each instance of this concept by dividing the book sales by the number of book authors.

To compute sales for each genre and country, we build a cube any cell of which is one genre and one country:

```
CUBE(Genres g, Countries c)
```

Each element of the result set returned by this query, called a *cell* in OLAP, is a pair of one genre and one country. Now we can compute sales for each cell in the query body as follows:

```
CUBE(Genres g, Countries c)
BODY {
```

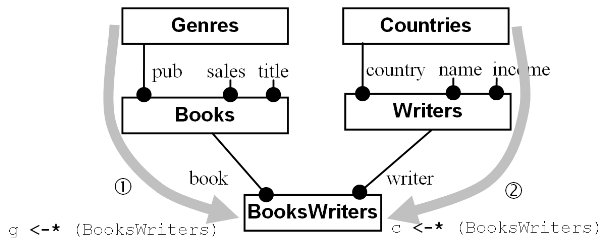


Figure 8. Multidimensional analysis in COM

```
      BW = g <-* (BooksWriters) AND
           c <-* (BooksWriters)
   }
   RETURN g.name, c.name, SUM(bw.sales)
```

Here `BW` is a cell and `BooksWriters` is a fact collection. A cell is computed as an intersection of all facts belonging to the current genre and all facts belonging to the current country. Then we simply return total sales for all facts in the cell where fact sales are computed in the derived method. Note that this approach does not use group-by operation which is replaced by de-projection.

5 Conclusion

The concept-oriented model is based on three general principles but intrinsically supports many patterns of thought used in other data models including set-based, hierarchical, multidimensional and semantic views on data. However, the largest overlap is with the object-based view and therefore COM can be characterized as taking its roots in object-orientation. In comparison with object-orientated approach, COM makes the following major contributions:

Concepts instead of classes. If class has only one constituent then concept combines two constituents in one construct: identity and entity. Data modeling is then reduced to describing identity-entity couples. This duality produces a nice yin-yang style of balance and symmetry between two orthogonal branches: identity modeling and entity modeling. In particular, this generalization allows us to model domain-specific identities instead of having only platform-specific ones. Concepts provide a basis for a new approach to type modeling. One its application is a unified mechanism for modeling relation types and domain types (which is one of the oldest problems in data modeling) by defining a domain as a set of identity-entity couples.

Inclusion instead of inheritance. Classical inheritance assumes that objects exist in one flat space where they are identified using one type of references. Inclusion generalizes this view by permitting objects to exist in a hierarchy where they are identified by hierarchical addresses. In this case both concepts and their instances exist within a hierarchy. This eliminates the asymmetry between classes defined as a hierarchy and their instances existing in flat space. Data modeling is then reduced to describing such hierarchical address space where objects are supposed to exist by focusing on identity modeling as opposed to traditional entity-centric approaches. This generalization turns objects into sets (of their children) and makes the whole approach inherently set-based rather than instance-based. Another important property of inclusion is that it retains all properties of classical inheritance and can be employed for reuse. Inheritance (IS-A) is revisited and considered a particular case of inclusion (IS-IN), that is, to put an object in a container means to inherit its properties.

Partial order instead of graph. Elements in COM are partially ordered where references represent greater elements and dimension types of concepts represent greater concepts. Data modeling is then reduced to ordering elements while other properties and mechanisms are derived from this relation. In particular, to be characterized by some property is equivalent to have another object as a greater element. Partial order also emphasizes the set-oriented nature of COM because elements are treated as sets of their lesser elements. Another important consequence of having partial order is that it allows us to implement an alternative approach to inference which is based on the data itself rather than on inference rules.

Due to this generality, COM decreases the existing mismatches between various kinds of models and methodologies:

Identity vs. entity and value vs. object. COM treats values and objects as two sides of one element by uniting identity modeling and entity modeling. In addition, COM provides an alternative approach for unifying domain and relational modeling.

Data modeling vs. programming. COM is integrated with a novel approach to programming, called concept-oriented programming (COP) [25, 30, 32] so that programming and data modeling are two branches of one methodology by decreasing the old and deeply rooted incongruity between these two branches of computer science [9, 8, 2]. In this context, COM can be defined as COP plus data semantics (implemented via partial order).

Transactional vs. analytical. COM unites transactional and analytical approaches to data modeling by narrowing the gap between operational systems and data warehouse (OLTP-OLAP impedance mismatch) by providing direct support for analytical operations which is currently a highly actual problem [23].

Logical vs. conceptual. COM is an inherently semantic data model which provides built-in mechanisms for reasoning about data by retaining conventional mechanisms for data access. In addition, COM essentially achieves the goals pursued by the nested relation model [14] and the universal relation models [15] but does it on the order-theoretic basis.

Instance-based vs. set-based. COM allows for both instance-level and set-level data access and manipulations.

Using the notions of concepts, inclusion and partial order, COM can describe a wide range of existing data modeling techniques and patterns. In particular, this approach does not use low level join and group-by operations. Taking into account its simplicity and generality, COM seems rather perspective direction for further research and development activities in the area of data modeling and object databases.

References

[1] S.Abiteboul, P.C.Kanellakis: Object identity as a query language primitive. *Journal of the ACM (JACM)*, 45(5): 798–842, 1998.

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

[3] 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, 1990.

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

[5] 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.

[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] W.R.Cook, A.H.Ibrahim: Integrating Programming Languages and Databases: What is the Problem? *ODBMS.ORG*, Expert Article, 2006.

[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. *Proc. Intl. Workshop on Object-Oriented Database Systems*, 1986, 2–4.

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

[12] P.Hall, J.Owlett, S.Todd: Relations and Entities. *Modeling in DataBase Management Systems*, G.M.Nijssen (Ed.), North Holland, 1976, 201–220.

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

[14] G.Jaeschke, H.J.Schek: Remarks On The Algebra Of Non First Normal Form Relations. *Proc. PODS'82*, 1982, 124–138.

[15] W.Kent, Consequences of assuming a universal relation. *ACM Transactions on Database Systems (TODS)*, 6(4): 539–556, 1981.

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

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

[18] G.M.Kuper, M.Y.Vardi: The Logical Data Model. *ACM Transactions on Database Systems*, 18(3): 379–413, 1993.

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

[20] J.Mylopoulos, Data Semantics Revisited: Databases and the Semantic Web, *DASFAA'04*, 2004.

[21] J.Peckham, F.Maryanski, Semantic data models. *ACM Computing Surveys (CSUR)*, 20(3): 153–189, 1988.

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

[23] H.Plattner: A common database approach for oltp and olap using an in-memory column database. *Proc. SIGMOD'09*, 2009, 1–2.

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

[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: Hierarchical Multidimensional Modeling in the Concept-Oriented Data Model. *Proc. 3rd Intl. Conference on Concept Lattices and Their Applications (CLA'05)*, 2005, 123–134.

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

[28] A.Savinov: Grouping and Aggregation in the Concept-Oriented Data Model. *Proc. 21st Annual ACM Symposium on Applied Computing (SAC'06)*, 2006, 482–486.

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

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

[31] A.Savinov: Concept-Oriented Model. *Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends*, V.E.Ferraggine, J.H.Doorn, L.C.Rivero (Eds.), IGI Global, 2009, 171–180.

[32] A.Savinov: Concept-Oriented Programming. *Encyclopedia of Information Science and Technology*, 2nd Edition, M.Khosrow-Pour (Ed.), IGI Global, 2009, 672–680.

[33] A.Savinov: Concept-Oriented Query Language for Data Modeling and Analysis. *Advanced Database Query Systems: Techniques,*

Applications and Technologies, L.Yan, Z.Ma (Eds.), IGI Global, 2011, 85–101.

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

[35] 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.

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

[37] D.C.Tsichritzis, F.H.Lochovsky: Hierarchical Data-Base Management: A Survey. *ACM Computing Surveys (CSUR)*, 8(1): 105–123, 1976.

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

Alexandr Savinov,

Received November 30, 2011

SAP Research Dresden,
SAP AG
Chemnitzer Str. 48,
01187 Dresden, Germany
E–mail: *alexandr.savinov@sap.com*
Home page: *http : //conceptoriented.org/savinov*