Transforming Conceptual Data Models into an Object Model*

Rudolf Herzig and Martin Gogolla

Technische Universität Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, W–3300 Braunschweig, Fed. Rep. Germany
e–mail: {herzig|gogolla}@idb.cs.tu-bs.de

Abstract. In this paper a conceptually simple structural object model focusing on object types, attributes and ISA relationships is introduced. The model is derived mainly from an extended Entity-Relationship approach, but concepts from other semantic and object-oriented models have influenced its features. It is shown how high-level conceptual data models can be mapped to this model, and to what extent the object model subsumes classical modeling paradigms.

1 Introduction

In recent years numerous data models for the conceptual modeling of information systems have been proposed. Among them there are semantic data models like SDM [13], IFO [3] and (extended) ER models [8, 9, 10, 24, 31, 34], complex object models like [1, 18, 26], purely object-based models like FDM [28] as well as object-based models with complex values such as [22]. All the approaches have different motivation, terminology, and aims. Here we aim to show you how some of these models can be translated into a conceptually simple object model (OBM), which can serve as a common basis for the constructs employed in those approaches.

Our object model is mathematically precisely defined, i.e., it is given an abstract syntax and formal semantics, and it is complete, since we gave semantics to the query and integrity specification component in [12], and to the data manipulation component in [14], by using a well-defined calculus.

As should become clear, the object model is not intended to be a tool for conceptual modeling. For this task one of the well-known semantic data models offering high-level modeling primitives should be used. The purpose of the object model is to provide a kind of target model for the logical design, i.e., the chosen design should be translated into it. In this respect the role of the object model should be seen analogously to the role played by the relational model to the logical design. However, although the object model is conceptually simpler than high-level semantic models, it is clearly much richer than the relational model. More importantly, it is still rich enough to apply results obtained in its simple context to more complex ones. For instance, the calculus defined for the object model may be applied to IFO in a straightforward manner.

The paper is organized as follows. In Sect. 2 a fictitious conceptual data model summarizing concepts of a number of semantic data models is introduced, and in Sect. 3 it is

* In: G. Pernul and A M. Tjoa, editors, *Proc. 11th Int. Conf. on the Entity-Relationship Approach*, pages 280–298. Springer LNCS 645, 1992
This work was funded by the German Ministry for Research and Technology (BMFT) under Grant No. 01 IS 203 D.

shown how this model can be mapped into our object model which is conceptually much simpler because it deals with atomic object types only. While Sect. 2 adopts a rather informal view, the object model is subject to rigorous mathematical semantics in Sect. 3. The object model is related to other modeling paradigms like complex object models and purely object-based models in Sect. 4. Finally, our observations and conclusions are given in the last section.

2 A fictitious conceptual data model

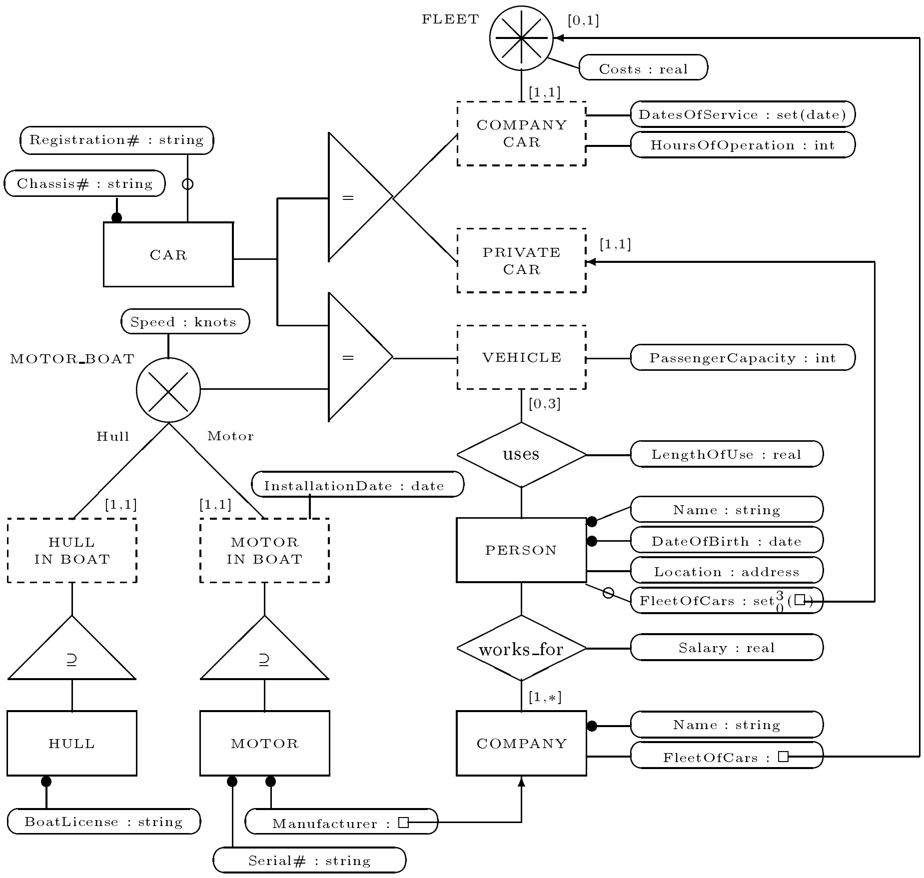


Fig. 1. A conceptual schema

Let us look at the conceptual schema in Fig. 1. The schema combines modeling primitives of IFO [3], the ER model [8], and an extended ER approach [10], covering the following fundamental data abstractions of the semantic data models (cf. [17, 25, 27]):

Object types are used to collect physical objects (entities) in the world such as `PERSON`s, `MOTOR_BOAT`s, or conceptual objects such as `COMPANY`s. This abstraction is called *classification* in the literature. Typically, a semantic data model provides means to specify *atomic types* and type constructors for building *complex types*.

– Atomic object types are usually distinguished either as being *printable* (literal or lexical object types, LOTs), e.g. `string`, or *abstract* (non-lexical object types, NOLOTs [33]), e.g. `PERSON`. In the following, printable object types are called "data types", whereas the notion "object type" is used for abstract object types only. In Fig. 1 object types (entity types) are depicted by rectangles.
– The symbols $\circledast$, $\otimes$ and $\diamond$ denote complex object types. To be more specific, we have a *set object type* (association [5], collection [18], grouping [13]) `FLEET`, a *tuple object type* (aggregation [29], composition [18]) `MOTOR_BOAT`, and the *relationship types* `uses` and `works_for`. Members of set object types are sets over an underlying type, whereas members of both tuple object types and relationship types are ordered n-tuples. Of course, relationships can be seen as (tuple) objects, and it would be possible to combine tuple object types and relationship types under one single heading, but this addresses an advanced level of abstraction[2].

Attributes given by ovals are used to model properties of objects. Several kinds of attributes can be identified.

– First we have *data-valued* attributes, such as `Name` : `PERSON` $\rightarrow$ `string`.
– In contrast, attributes ranging over abstract object types, such as `Manufacturer` : `MOTOR` $\rightarrow$ `COMPANY`, are called *object-valued*.
– Both data- and object-valued attributes are allowed to be *multi-valued*, e.g., `FleetOfCars` : `PERSON` $\rightarrow$ `set(PRIVATE_CAR)`.

ISA relationships form the third fundamental component of semantic data models. In Fig. 1 ISA relationships are represented by triangles which we call **distributors**. Object types on the base line of a distributor are called input types, and object types on the pointed side are called output types. Distributors take objects of the input types and associate them with output types. The distribution is "lax" in the sense that there may be objects in the input types, which are not associated with any output type.

If an input object is associated with an output type, this brings about a role object or *aspect* of this object to be created. All objects in the output types originate from this mechanism. Therefore, the output types of distributors are also called **ISA-dependent types**. In contrast, object types which do not appear on the output side of any distributor are called **base types**. We assume that there are no cyclic ISA relationships. Consequently, we have for every ISA-dependent object an underlying base object. In Fig. 1 ISA-dependent object types are represented as base object types except for a dashed line as boundary of the rectangle.

Depending on the number of the input types and the output types of a distributor, two special kinds of ISA relationships can be ascertained, namely *specialization* and *categorization*:

– If there is only input type, then specializations, such as the specialization of a `MOTOR` to a `MOTOR_IN_BOAT`, are modeled. If there are several output types at the same time, we have *disjoint specializations*, such as `CAR` to `COMPANY_CAR` and `PRIVATE_CAR`.
– If there is only output type, then categorizations [9] are modeled. For example, both `STUDENT`s and `TEACHER`s may act as `BORROWER`s in a lending library. By an additional constraint (see below), categorizations can be extended to generalizations by demanding that each input object of a distributor must be assigned to the output type, as in the case of generalizing `CAR`s and `MOTOR_BOAT`s to `VEHICLE`s.

[2] A similar distinction was made in [34] by the notion of *abstract* and *associative* relationships.

The concepts specialization and categorization overlap if there is only one object type on each side of a distributor.

To summarize the basic modeling primitives we maintain that object types are the fundamental building blocks of a semantic data model schema, and that there are three ways to express relationships between the different object types: to relate objects we can build complex object types, or make use of attributes modeling functional relationships, or distribute objects to ISA-dependent types.

Up to now we have described the basic modeling primitives. The schema in Fig. 1 includes some further features which can be characterized as *structural restrictions*.

Object participation constraints: The participation of objects in complex objects can be limited by lower and upper bounds, e.g., every `MOTOR_IN_BOAT` must be installed in exactly one `MOTOR_BOAT`, every `COMPANY_CAR` must appear in exactly one `FLEET`, for every `COMPANY` there must be at least one `PERSON` working for it, and each `VEHICLE` must not be used by more than three `PERSON`s. Object participation constraints are used in many semantic data models, e.g. IRIS [11], in particular they cover cardinality constraints and existence dependencies in ER models [9].

Cardinalities wrt. multi-valued attributes: The cardinality of a set appearing as a value of a multi-valued attribute can be restricted by lower and upper bounds, too, e.g., every `PERSON` must not own more than three `PRIVATE_CAR`s given the attribute `FleetOfCars` (or more precisely `PERSON.FleetOfCars`).

Cardinalities wrt. ranges of attributes: In some cases we need cardinalities even for the ranges of attributes. In our example it is not permitted for a `FLEET` to belong to more than one `COMPANY` via the attribute `FleetOfCars`, and a `PRIVATE_CAR` must belong to exactly one `PERSON`. Hence `COMPANY.FleetOfCars` can be characterized as a *non-overlapping* attribute, and `PERSON.FleetOfCars` is non-overlapping and moreover *exhaustive*. Both attribute restrictions are known from SDM [13], for example.

Mandatory attributes: We assume that the interpretations of all object types include a special element $\perp$ representing the *undefined* or *null value*. Consequently, all attributes are *optional* by default. The optionality can be excluded by a constraint. In the example schema optional attributes are distinguished from mandatory attributes by a circle ($\circ$), i.e., `Registration#` and `PERSON.FleetOfCars` are optional attributes, whereas all other attributes are mandatory.

Bi-directional ISA relationships: In general, triangles represent ISA relationships directed from the output side to the input side (every `MOTOR_IN_BOAT` is a `MOTOR`, but a `MOTOR` is not necessarily a `MOTOR_IN_BOAT`). To express that an ISA relationship also holds for the other direction, as it is in case of *generalizations* (every `CAR` and every `MOTOR_BOAT` is a `VEHICLE`) and *partitions* (i.e. covering disjoint specializations) (every `CAR` is either a `COMPANY_CAR` or a `PRIVATE_CAR`), the corresponding triangle can be marked by '='.

Key concept: The identity of abstract objects is not visible to the user. But especially with respect to updates it is necessary that the user is able to refer to a distinct object. Therefore, we have for each base object type at least one key attribute denoted by a thick dot ($\bullet$). The key integrity assures that each base object can be identified uniquely by the values of its key attributes. For complex object types and ISA-dependent types we do not need a key concept, because the identity of each complex object is already given by the construction of the object, and an ISA-dependent object inherits its identity from the underlying base object.

In the next section the conceptual schema of Fig. 1 is mapped into a simple formal data model. Provided that this model finds an implementation, this might be considered as a logical design step within database design process [7, 30].

3 Formal Definition of an Object Schema

In this section we present the basic terms relevant to our notion of an object schema. We distinguish between a data level given by a *data signature* DS and an object level given by an *object schema* over DS. The data signature contains a finite set of *data types* (or printable object types) together with a set of operations and predicates defined upon them. In contrast to (abstract) object types the semantics of data types is fixed. From a semantic point of view, the data level is the kernel of the object level because abstract objects can be observed only by means of data types used as attribute domains.

For all terms discussed in this section we give precise mathematical definitions employing mainly the notions set and function.

Axiomatic conventions. Let $|\mathbf{SET}|$ denote the class of sets, $|\mathbf{FISET}|$ the class of finite sets, $|\mathbf{FUN}|$ the class of total functions and $|\mathbf{REL}|$ the class of relations. There are the obvious inclusions $|\mathrm{FISET}| \subseteq |\mathrm{SET}|$ and $|\mathrm{FUN}| \subseteq |\mathrm{REL}| \subseteq |\mathrm{SET}|$. Assume sets $\mathrm{S}, \mathrm{S}_1, \ldots, \mathrm{S}_n \in |\mathrm{SET}|$ are given. Then $\boldsymbol{F}(\mathbf{S})$ denotes the restriction of the powerset $\boldsymbol{P}(\mathbf{S})$ of S to finite sets, $\mathbf{S}^*$ the set of finite lists over S, $\mathbf{S}^+$ the set of finite non-empty lists over S, and $\mathbf{S_1} \times \ldots \times \mathbf{S_n}$ the Cartesian product of the sets $\mathrm{S}_1,\ldots,\mathrm{S}_n$. Finite sets are written as $\{\mathbf{c_1}, \ldots, \mathbf{c_n}\}$, lists as $\langle \mathbf{c_1}, \ldots, \mathbf{c_n} \rangle$, and elements of the Cartesian product as $(\mathbf{c_1}, \ldots, \mathbf{c_n})$.

3.1 Data Level

The syntax of a data signature gives the names of the data sorts and the operations and predicates upon them. We assume certain sorts (e.g., `int`, `real`, `string`) and standard operations (e.g., $+_{\mathtt{int}} : \mathtt{int} \times \mathtt{int} \rightarrow \mathtt{int}$) and predicates (e.g., $\leq_{\mathtt{int}}$: $\mathtt{int} \times \mathtt{int}$) to be predefined. The semantics of a data signature associates a (possibly infinite) set with every sort name, and functions and relations with the operation and predicate names. This interpretation will be fixed once and for all and will not change. In the following formal definition we shall only consider names of data types, i.e., data sorts. A complete definition of a data signature, including data operations and predicates, can be found in [12].

Definition 1 (data sorts). The **syntax of data sorts** is given by a finite set DATASORT. The **semantics of data sorts** is given by the total function $\mu[\mathrm{DATASORT}]$: $\mathrm{DATASORT} \rightarrow |\mathrm{SET}|$ such that $d \in \mathrm{DATASORT}$ implies $\perp \in \mu[\mathrm{DATASORT}](d)$. □

Predefined sorts like `int`, `real`, `string` have the 'usual' semantics, e.g., $\mu[\mathrm{DATASORT}](\mathtt{int}) = \mathbb{Z} \cup \{\perp\}$. The element $\perp$ denotes the 'undefined' value. We require every data sort d (and later every object type o) to contain $\perp$, because it is necessary to have such a value as result of incorrect applications of operations, e.g., the division by zero always results in $\perp$. We assume the interpretations of all data sorts (and later object types) to be disjoint, which is partially achieved by including the name or token of a type into the structure of values.

Besides the standard applications, the data signature may contain user-defined sorts, operations and predicates. With respect to user-defined sorts, atomic data types (e.g. enumeration types such as `color` = {green, yellow, red}) are possible as well as arbitrary complex data types definable by *sort expressions.*

Definition 2 (sort expressions). Let a set S $\in$ |SET| together with a (semantic) function $\mu[\mathrm{S}] : \mathrm{S} \to |\mathrm{SET}|$ (such that $\perp \in \mu[\mathrm{S}](s)$ for every $s \in \mathrm{S}$) and a set A of (attribute) names be given. The **syntax of sort expressions over S** is given by the set EXPR(S) determined by the following rules.

(i) If $s \in \mathrm{S}$, then $\boldsymbol{s} \in \mathrm{EXPR(S)}$.
(ii) If $s \in \mathrm{EXPR(S)}$, then $\mathbf{set}(\boldsymbol{s}) \in \mathrm{EXPR(S)}$.
(iii) If $s_1, \ldots, s_n \in \mathrm{EXPR(S)}$ and $a_1, \ldots, a_n \in \mathrm{A}$ with $a_i \neq a_j$ for $i \neq j$, then $\mathbf{tuple}(\boldsymbol{a_1{:}s_1,\ldots,a_n{:}s_n}) \in \mathrm{EXPR(S)}$.
(iv) If $s_1, \ldots, s_n \in \mathrm{EXPR(S)}$ and $a_1, \ldots, a_n \in \mathrm{A}$ with $a_i \neq a_j$ for $i \neq j$, then $\mathbf{union}(\boldsymbol{a_1{:}s_1,\ldots,a_n{:}s_n}) \in \mathrm{EXPR(S)}$.

The **semantics of sort expressions over S** is a total function $\mu[\mathrm{EXPR(S)}] : \mathrm{EXPR(S)} \to |\mathrm{SET}|$ determined by the following rules.

(i) $\mu[\mathrm{EXPR(S)}](s) := \mu[\mathrm{S}](s)$.
(ii) $\mu[\mathrm{EXPR(S)}](\mathrm{set}(s)) := F(\mu[\mathrm{EXPR(S)}](s)) \cup \{\perp\}$.
(iii) $\mu[\mathrm{EXPR(S)}](\mathrm{tuple}(a_1{:}s_1, \ldots, a_n{:}s_n)) :=$
$(\{a_1{:}\underline{s}_1 \mid \underline{s}_1 \in \mu[\mathrm{EXPR(S)}](s_1)\} \times \ldots \times \{a_n{:}\underline{s}_n \mid \underline{s}_n \in \mu[\mathrm{EXPR(S)}](s_n)\}) \cup \{\perp\}$.
(iv) $\mu[\mathrm{EXPR(S)}](\mathrm{union}(a_1{:}s_1, \ldots, a_n{:}s_n)) :=$
$\{a_1{:}\underline{s}_1 \mid \underline{s}_1 \in \mu[\mathrm{EXPR(S)}](s_1)\} \cup \ldots \cup \{a_n{:}\underline{s}_n \mid \underline{s}_n \in \mu[\mathrm{EXPR(S)}](s_n)\} \cup \{\perp\}$. □

A complex data type `graph` is constructed by `tuple(N:nodes, E:edges)` in which `nodes` is given by `set(node)`, `edges` by `set(tuple(Start:node, End:node))`, and `node` is an enumeration type. For instance, (`N`:{ 1,2 } , `E`:{ (`Start`:1, `End`:2), (`Start`:1, `End`:1) }) is a proper value of type `graph`. The constructor `union` allows the modeling of alternatives. For example, a data type `amount` could be given by `union(Dollar:int, Sterling:int)`, and `Dollar`:10 or `Sterling`:5 are possible values of type `amount`. Besides the proposed further type constructors like `list` or `bag` (multiset) are possible.

Sort expressions induce a number of **aggregate functions** like `CNT` counting the elements in a set, `IN` representing the element-of relation, operations selecting components of a tuple or alternatives of a union. and many other operations and predicates. For details we refer to [12].

3.2 Object Level

The object schema is the central notion of this section. The syntax of such a schema introduces names for object types, for distributors which make some object types to ISA-dependent types, and for attributes of objects. The semantics of an object schema associates sets and functions with these items. For instance, for every object type there is a (finite) set of current objects, and for every attribute name there is a function giving the current value of the attribute for every object. These sets and functions constitute a (database) state, which can be altered in the course of time (in contrast to the interpretation of the data types, which is fixed). The connection to the data signature is established via data-valued attributes.

Definition 3 (object schema). Let a data signature DS be given. The **syntax of an object schema OBM(DS) over DS** is given by

- the finite sets BASE_OBJTYPE, ISA_OBJTYPE, ATTR, DISTR,
- and the total functions

$$\begin{array}{lll} \mathit{source}: & \text{ATTR} & \to \text{OBJTYPE}, \\ \mathit{destination}: & \text{ATTR} & \to \text{EXPR(SORT)}, \\ \mathit{parent\_distr}: & \text{ISA\_OBJTYPE} & \to \text{DISTR}, \\ \mathit{input}: & \text{DISTR} & \to F(\text{OBJTYPE}) - \{\emptyset\}, \\ \mathit{key}: & \text{BASE\_OBJTYPE} & \to \text{ATTR}^+. \end{array}$$

OBJTYPE = BASE_OBJTYPE $\cup$ ISA_OBJTYPE refers to the set of all object types and SORT = DATASORT $\cup$ OBJTYPE to the set of all basic sorts. The function *output* : DISTR $\to F$(ISA_OBJTYPE) is defined by $output(d) := \{\, o \mid parent\_distr(o) = d \,\}$.
If $a \in$ ATTR with $source(a) = o$ and $destination(a) = s$, this is denoted as $a : o \to s$. If $d \in$ DISTR with $input(d) = \{i_1, \ldots, i_m\}$ and $output(d) = \{o_1, \ldots, o_n\}$, this is denoted as $d(i_1, \ldots, i_m \supseteq o_1, \ldots, o_n)$.
There are some syntactical restrictions:

- For $key(o) = \langle a_1, \ldots, a_n \rangle$ there is $a_i \neq a_j$ for $i \neq j$, and $a \in key(o)$ implies $source(a) = o$.
- The function *parent_distr* has to be surjective.

($*$) It is not permitted for $obj\_dep^+(o, o)$ to hold for some $o \in$ OBJTYPE, where obj_dep^+ is the transitive closure of the relation *obj_dep* defined by: if $o_1 \in$ ISA_OBJTYPE and $o_2 \in input(parent\_distr(o_1))$ or $o_1 \in$ BASE_OBJTYPE and there is a key attribute $a \in key(o_1)$ with a : $o_1 \to s$ and o_2 appears in the construction of the sort expression s, then $obj\_dep(o_1, o_2)$ holds.

The **semantics of an object schema OBM(DS) over DS** is given by the total functions

- μ[BASE_OBJTYPE] : BASE_OBJTYPE $\to$ |FISET| such that $o \in$ BASE_OBJTYPE implies $\perp \in \mu$[BASE_OBJTYPE](o),
- μ[DISTR] : DISTR $\to$ |FUN| such that $d(i_1, \ldots, i_m \supseteq o_1, \ldots, o_n)$ implies $\mu[\text{DISTR}](d) : \bigcup_{k=1}^{m} \mu[\text{OBJTYPE}](i_k) \to \{o_1, \ldots, o_n\} \cup \{\perp\}$,
- μ[ISA_OBJTYPE] : ISA_OBJTYPE $\to$ |FISET| such that $o \in$ ISA_OBJTYPE implies μ[ISA_OBJTYPE]$(o) :=$

$$\begin{array}{l} \{\, \mathrm{o}{:}\underline{b} \mid b \in \text{BASE\_OBJTYPE and } \underline{b} \in \mu[\text{BASE\_OBJTYPE}](b) \text{ and} \\ \quad \text{there are } o_1, \ldots, o_n \in \text{ISA\_OBJTYPE with } o_n = o \text{ such that} \\ \qquad \mu[\text{DISTR}](\mathit{parent\_distr}(o_1))(\underline{b}) = o_1 \text{ and} \\ \qquad \mu[\text{DISTR}](\mathit{parent\_distr}(o_2))(\mathrm{o}_1{:}\underline{b}) = o_2 \text{ and} \\ \qquad \vdots \\ \qquad \mu[\text{DISTR}](\mathit{parent\_distr}(o_n))(\mathrm{o}_{n-1}{:}\underline{b}) = o_n \,\} \cup \{\perp\}, \end{array}$$

- μ[ATTR] : ATTR $\to$ |FUN| such that a : $o \to$ s implies μ[ATTR](a) : μ[OBJTYPE]$(o) \to \mu$[EXPR(SORT)](s), and
  - if $o \in$ BASE_OBJTYPE and a $\in key(o)$, then μ[ATTR](a)$(\underline{o}) = \perp$ implies $\underline{o} = \perp$.

For $o \in$ BASE_OBJTYPE, $key(o) = \langle a_1, \ldots, a_n \rangle$, $a_i : o \rightarrow s_i$ $(i = 1, \ldots, n)$ the function $id_o : \mu[\text{BASE\_OBJTYPE}](o) \rightarrow \mu[\text{EXPR(SORT)}](\text{tuple}(a_1{:}s_1,\ldots,a_n{:}s_n))$ is defined by $id_o(\underline{o}) := (a_1{:}\mu[\text{ATTR}](a_1)(\underline{o}), \ldots, a_n{:}\mu[\text{ATTR}](a_n)(\underline{o}))$ iff $\underline{o} \neq \bot$, and $id_o(\bot) = \bot$. **Key integrity:** $\underline{o}_1, \underline{o}_2 \in \mu[\text{BASE\_OBJTYPE}](o)$ with $id_o(\underline{o}_1) = id_o(\underline{o}_2)$ implies $\underline{o}_1 = \underline{o}_2$.

μ[OBJTYPE] and μ[SORT] are determined by the corresponding $\mu's$ on the right-hand side of the equations above. For all $\mathrm{s}_1, \mathrm{s}_2 \in$ SORT there is $\mu[\text{SORT}](\mathrm{s}_1) \cap \mu[\text{SORT}](\mathrm{s}_2) = \{\bot\}$. Every function μ[ATTR](a) and μ[DISTR](d) has to preserve the undefined value, e.g., $\mu[\text{ATTR}](\mathrm{a})(\bot) = \bot$, and so on. □

Let us now consider the textual object schema in Fig. 2 belonging to the conceptual schema in Fig. 1. The schema is denoted in a linear way. Specifications enclosed in `<   >` contain information about the logical object identity, i.e., the key group for base object types and the parent distributor for ISA-dependent object types. Please note that specifications starting with '`/`' are to be ignored at this stage. The syntax of the object schema is given by the sets:

BASE_OBJTYPE	= { `CAR`, `HULL`, ..., `FLEET`, `uses`, `works_for` }
DISTR	= { `specialization_HULL`, ..., `generalization_VEHICLE` }
ISA_OBJTYPE	= { `HULL_IN_BOAT`, `MOTOR_IN_BOAT`, ..., `VEHICLE` }
ATTR	= { `Chassis#`, `Registration#`, ..., `PassengerCapacity` }

We assume that attributes of the same name defined on different object types remain distinguishable by using the object type as a prefix, e.g., `PERSON.Name` and `COMPANY.Name`.

The elements of BASE_OBJTYPE are called **base object types**. With respect to the example of Fig. 1 the notion "base object type" includes (base) entity types , set object types $\circledast$, tuple object types $\otimes$, and relationship types $\diamond$. The set of actual objects belonging to an object type o is given by μ[BASE_OBJTYPE](o).

$\mu[\text{BASE\_OBJTYPE}](\texttt{CAR}) = \{\underline{c}_1, \ldots, \underline{c}_m\} \cup \{\bot\}$
$\mu[\text{BASE\_OBJTYPE}](\texttt{MOTOR\_BOAT}) = \{\underline{mb}_1, \ldots, \underline{mb}_n\} \cup \{\bot\}$

Throughout the paper the underbar notation will always indicate a value of semantic domain. For base object types, the values $\underline{c}_i$, $\underline{mb}_j$, etc. represent *abstract object identifiers* (OIDs, surrogate keys), which belong to real world entities. We are not going into the subject of the structure of object identifiers, because we assume that object identifiers are internal values not visible to the user.

The elements of DISTR are called **distributors**; they are represented by triangles in Fig. 1. A distributor d may be regarded as a rearrangement of object types associating with every object in the input types an output type. The distribution is "lax" in the sense that there might be objects in the input types appearing in none of the output types if they are mapped to the 'undefined type'.

$\mu[\text{DISTR}](\texttt{partition\_CAR}) : \{\underline{c}_1, \ldots, \underline{c}_m\} \cup \{\bot\} \rightarrow \{\texttt{COMPANY\_CAR}, \texttt{PRIVATE\_CAR}\} \cup \{\bot\}$
$\mu[\text{DISTR}](\texttt{generalization\_VEHICLE}) : \{\underline{c}_1, \ldots, \underline{c}_m\} \cup \{\underline{mb}_1, \ldots, \underline{mb}_n\} \cup \{\bot\}$
$\rightarrow \{\texttt{VEHICLE}\} \cup \{\bot\}$

In contrast to base object types, the semantics of object types on the output side of distributors are fixed by the semantics of the "parent" distributor. Therefore, the output types given by the set ISA_OBJTYPE are called **ISA-dependent object types**. In Fig. 1 ISA-dependent object types are represented by a dashed box .

```
schema example
  base_objtypes
    CAR           < Chassis# : string >
                  Registration# : string /optional;
    HULL          < BoatLicense : string >;
    MOTOR         < Manufacturer : COMPANY, Serial# : string >;
    PERSON        < Name : string, DateOfBirth : date >
                  Location : address,
                  FleetOfCars : set(PRIVATE_CAR)
                                /optional /multi[0,3] /inv[1,1];
    COMPANY       < Name : string >
                  FleetOfCars : FLEET /inv[0,1];
    MOTOR_BOAT    < Hull  : HULL_IN_BOAT /inv[1,1],
                    Motor : MOTOR_IN_BOAT /inv[1,1] >
                  Speed : knots;
    FLEET         < Members : set(COMPANY_CAR) /inv[1,1] >
                  Costs : real;
    uses          < P : PERSON, V : VEHICLE /inv[0,3] >
                  LengthOfUse : real;
    works_for     < P : PERSON, C : COMPANY /inv[1,*] >
                  Salary : real;

  isa_objtypes
    HULL_IN_BOAT  < specialization_HULL >;
    MOTOR_IN_BOAT < specialization_MOTOR >
                  InstallationDate : date;
    COMPANY_CAR   < partition_CAR >
                  DatesOfService   : set(date),
                  HoursOfOperation : int;
    PRIVATE_CAR   < partition_CAR >;
    VEHICLE       < generalization_VEHICLE >
                  PassengerCapacity : int;

  distributors
    specialization_HULL     HULL;
    specialization_MOTOR    MOTOR;
    partition_CAR           CAR /strict;
    generalization_VEHICLE  CAR, MOTOR_BOAT /strict;
```

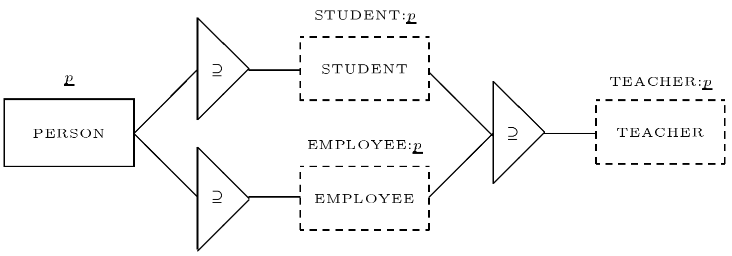
Fig. 2. Object schema

$$\mu[\text{ISA\_OBJTYPE}](\texttt{COMPANY\_CAR}) = \{\ \texttt{COMPANY\_CAR}{:}\underline{c} \mid \underline{c} \in \{\underline{c}_1, \ldots, \underline{c}_m\} \text{ and } \mu[\text{DISTR}](\texttt{partition\_CAR})(\underline{c}) = \texttt{COMPANY\_CAR}\ \}$$
$$\mu[\text{ISA\_OBJTYPE}](\texttt{VEHICLE}) = \{\ \texttt{VEHICLE}{:}\underline{o} \mid \underline{o} \in \{\underline{c}_1, \ldots, \underline{c}_m\} \cup \{\underline{mb}_1, \ldots, \underline{mb}_n\} \text{ and } \mu[\text{DISTR}](\texttt{generalization\_VEHICLE})(\underline{o}) = \texttt{VEHICLE}\ \}$$

We did not formulate the semantics of ISA-dependent object types in such a way that $\mu[\text{ISA\_OBJTYPE}](o) := \{\ \underline{i} \mid \mu[\text{DISTR}](d)(\underline{i}) = o\ \}$, because we want the interpretations of all object types to be disjoint (with the exception of $\perp$). The acyclity of the object type dependency graph discussed below guarantees that there are no cycles in ISA relationships. In consequence, for every ISA-dependent object $\underline{o}$ (of type o) there will always be a base object $\underline{b}$ and n ISA-dependent object types o_i with $parent\_distr(o_i) = d_i$ $(i = 1, \ldots, n)$ so that

$$\underline{b} \overset{\mu(d_1)}{\longrightarrow} o_1; \quad (o_1{:}\underline{b}) \overset{\mu(d_2)}{\longrightarrow} o_2; \quad \cdots; \quad (o_{n-1}{:}\underline{b}) \overset{\mu(d_n)}{\longrightarrow} o_n; \quad (o_n{:}\underline{b}) = (o{:}\underline{b}) = \underline{o} \quad (n \geq 1) \ .$$

The object $\underline{b}$ is called the underlying base object or *ISA root* of $\underline{o}$. $o{:}\underline{b}$ could be read as "base object $\underline{b}$ playing role o". Formally, $o{:}\underline{b}$ establishes a new object distinguishable from $\underline{b}$, but in the data manipulation component of the object model inheritance can be supported in both directions (!) by implicit type conversions (see [12]). In an ISA hierarchy it is possible for there to be several derivation paths for a single ISA-dependent object:



For example, the ISA-dependent object `TEACHER`:$\underline{p}$ comes already into being if one of the objects `STUDENT`:$\underline{p}$ or `EMPLOYEE`:$\underline{p}$ is distributed to `TEACHER`.

Remark. The semantics chosen for ISA-dependent types is non-standard. To preserve disjointness of types, we particularly refrained from employing subtypes to describe ISA relationships. In our opinion a role object or *aspect* should be strictly disassociated from its origin, because a role object can show its own observation and its own behavior. Nevertheless, the dependency of an aspect on its parent object remains still reflected within the concepts of our approach. IFO, for example, takes the other view. In an IFO schema, ISA-dependent types are represented as *free nodes*. The *derived type* of a free node is determined by specialization and generalization edges. Applied to our example schema of Fig. 1, the derived type of `HULL_IN_BOAT` is `HULL`, and the derived type of `VEHICLE` is $\oplus$(`CAR`, $\otimes$(`HULL`, `MOTOR`)), i.e., instances of `VEHICLE` would be either instances of `CAR`, or composite objects built from instances of `HULL` and `MOTOR`.

The elements of ATTR are the **attribute names**. In Fig. 1 attributes are depicted by ovals. μ[ATTR](a) gives the attribute values of objects.

$$\mu[\text{ATTR}](\texttt{Chassis\#}) : \{\underline{c}_1, \ldots, \underline{c}_m\} \cup \{\perp\} \rightarrow \text{A}^* \cup \{\perp\} \ (= \mu[\text{DATASORT}](\texttt{string}))$$

The ranges of attributes can be given by arbitrary complex sort expressions. Attributes can relate to data or object sorts (single-valued attributes), and second tuple-, set- or union-valued attributes are possible. Deeper nesting allows the representation of *context-dependent* information [16]. Please note that components of tuple expressions are accessible by (attribute) names. Hence, attributes mapping to complex sort expressions relate to *complex attributes* as proposed in [24], for example.

We mentioned that the identity of base objects is invisible to the user. Hence, we can speak of an *abstract identity* of objects. It is difficult to obtain a specific base object, especially with respect to updates on objects. When such an update is to be carried out, the target object has to be described associatively through some related objects or values. However, there might be situations in which an update can only be performed as an update on sets, because there is no way to reference a distinct object. In order to facilitate object access, we offer a **key concept** providing base objects with an additional *logical identity*. One or more attributes of a base object type can be put together to form a *key group* of the object type, e.g., $key(\texttt{MOTOR}) = \langle\ \texttt{Manufacturer}, \texttt{Serial\#}\ \rangle$. Every key group creates an object generating function (or fetch operation) for the associated base object type, being $\texttt{MOTOR}^{\mathrm{GEN}} : \texttt{COMPANY} \times \texttt{string} \rightarrow \texttt{MOTOR}$ in our example. The key integrity ensures that this object generating operation is well-defined. The function returns $\perp$ if there is no abstract object with the given key values. In principle, we could assign more than one key group to a base object type achieving several logical access paths for abstract objects, as it has been proposed in SDM [13], but for simplicity we are not considering this aspect in this paper.

The key concept plays a crucial role in case of complex object types which are to be mapped into an object schema, because it allows the **simulation of complex object types** by atomic ones without losing the natural identity of composed objects. In more precise terms we are replacing *internal structure* by *attribute structure* [16], but we are retaining the means to express internal structure by keys. So, the aggregation type `MOTOR_BOAT` is modeled by turning the roles `Hull` and `Motor` to single-valued key attributes, and the association type `FLEET` is mapped to an atomic type with a multi-valued key attribute `members` denoting the members of a fleet. From a logical (user) point of view, nothing changes. In fact, the natural identity of constructed objects is split up to an *abstract identity* given by an object identifier (e.g., $\underline{mb}$ for a `MOTOR_BOAT`), and a *logical identity* given by the key attributes (e.g., $id_{\texttt{MOTOR\_BOAT}}(\underline{mb}) = (\texttt{Hull} : \underline{h}, \texttt{Motor} : \underline{m})$, in which $\underline{h}$ represents a `HULL` and $\underline{m}$ a `MOTOR`).

ISA relationships and object-valued key attributes lead to certain *object dependencies* in an object schema (see Fig. 3). The global structure of an object schema is restricted by condition ($*$) in the definition of an object schema demanding that there are no cycles in the object type dependency graph.

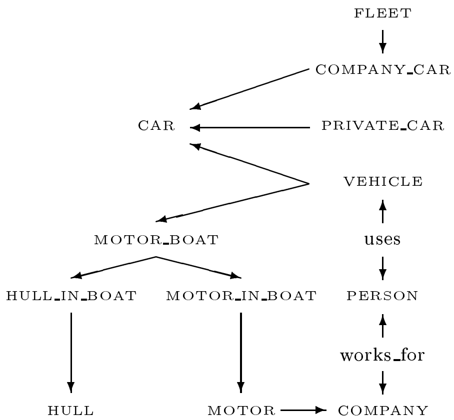


Fig. 3. Object type dependency graph

In summary we have SCHEMA=(BASE_OBJTYPE, ISA_OBJTYPE, ATTR, DISTR), and μ[SCHEMA] = (μ[BASE_OBJTYPE], μ[ISA_OBJTYPE], μ[ATTR], μ[DISTR]). The semantics of an object schema μ[SCHEMA] is also called a **state** σ belonging to the schema. We define $\boldsymbol{\pi}$**[SCHEMA]** as the set of all **possible states** Σ. State transitions between possible states are described by *elementary update operations*.

Apart from the fact that the object model includes the key concept as a base modeling primitive, the object model can be seen as a submodel of an **E**xtended **E**ntity-**R**elationship approach [10]. For this model a well-defined **EER calculus**, taking into account data operations on arbitrary user-defined data sorts and aggregate functions, was described in [15, 12]. By including object generating operations this calculus can be applied to the object model. Hence, the query component of the object model is well-understood. In addition, the calculus can be adopted to specify constraints on database states.

3.3 Constraints on Database States

The formal object schema only includes the basic modeling primitives as there are object types, attributes and ISA relationships represented by distributors. The one and only structural restriction which is covered by a formal object schema is the key concept. This is necessary, because we want all abstract objects to be accessible by printable data in every possible database state. However, there are further structural restrictions specified in Fig. 2 by expressions beginning with '/' which are yet not covered by the object schema. These structural restrictions can be regarded as *embedded constraints* (cf. [34]), because they could be included in the formal definition in a straightforward manner. We did not do so, because in the first place we wanted to keep the formal definition simple. Secondly, when structural restrictions are integrated in the formal definition, we also have to ask if these restrictions are consistent, i.e., if possible schema instances exist which satisfy all constraints. Last but not least structural restrictions directly complicate the definition of elementary update operations to database states, i.e., even elementary updates will sometimes escalate into transactions. For example, the insertion of a `MOTOR_BOAT` requires the specialization of a `HULL` to a `HULL_IN_BOAT`, the specialization of a `MOTOR` to a `MOTOR_IN_BOAT`, the insertion of a `MOTOR_BOAT`, and the generalization of the `MOTOR_BOAT` to a `VEHICLE`, together with the specification of the attributes `InstallationDate`, `Speed`, and `PassengerCapacity`, to satisfy all embedded constraints.

In our approach we view embedded constraints as a simple form of general **explicit integrity constraints**, which can be described by using formulas from the above mentioned calculus. For example, structural restrictions given in Fig. 2 can also be expressed by the following formulas.

- Object participation constraints `/inv`[min, max]:
  $\forall$`(mib:MOTOR_IN_BOAT)` `CNT`$\{\!|$ `mb` $|$ `(mb:MOTOR_BOAT)` $\wedge$ `Motor(mb)=mib` $|\!\}$ `= 1`
  $\forall$`(cp:COMPANY)` `CNT`$\{\!|$ `wf` $|$ `(wf:works_for)` $\wedge$ `C(wf)=cp` $|\!\}$ $\geq$ `1`
  $\forall$`(f:FLEET)` `CNT`$\{\!|$ `cp` $|$ `(cp:COMPANY)` $\wedge$ `FleetOfCars(cp)=f` $|\!\}$ $\leq$ `1`
- Cardinalities wrt. multi-valued attributes `/multi`[min, max]:
  $\forall$`(p:PERSON)` `DEF(FleetOfCars(p))` $\Rightarrow$ `CNT(FleetOfCars(p))` $\leq$ `3`
- Mandatory attributes (not `/optional`):
  $\forall$`(mib:MOTOR_IN_BOAT)` `DEF(InstallationDate(mib))`
- Bi-directional ISA relationships `/strict`:
  $\forall$`(c:CAR)` `DEF(COMPANY_CAR(c))` $\vee$ `DEF(PRIVATE_CAR(c))`
  $\forall$`(c:CAR)` `DEF(VEHICLE(c))`

Similar formulas may be used to describe more complex constraints, for example, covering constraints for ISA-dependent types, disjoint/covering/partition constraints across the object participation in multiple relationship types (see [34]), object participation constraints addressing more than one coordinate at a time [23], or general computable constraints [17]. By means of (static) explicit integrity constraints the set of all possible states

Σ is reduced to a set of **consistent states** Σ_{C}. Transitions between consistent database states are described by *complex update operations* (transactions) being generally sequences of elementary update operations.

At the end of this section, in Table 1 various semantic data models are described according to some of their structural aspects. Most of those aspects have been mentioned above. For details on comparison criteria for semantic data models consult [16, 17, 25, 27, 32], for example. We are now leaving semantic data models and turning to data models which are simpler, by concentrating on only a few modeling constructs. For most of these models there are actual implementations. Hence they can be considered as models for logical design.

					extended ER models				
	ER [8]	FDM [28]	SDM [13]	IFO [3]	ECR [9]	ERC [24]	ORAC [34]	EER [10]	OBM
Objects:									
- key concept	emb		$\surd$		emb	emb		emb	$\surd$
- set object types			$\surd$	$\surd$					
- tuple object types				$\surd$			$\surd$		
- relationship types	$\surd$				$\surd$	$\surd$	$\surd$	$\surd$	
Attributes:									
- single-valued	$\surd$	$\surd$	$\surd$	$\surd$	emb	emb	emb	$\surd$	$\surd$
- multi-valued		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
- object-valued		$\surd$	$\surd$	$\surd$				$\surd$	$\surd$
- complex				$\surd$		$\surd$	$\surd$	$\surd$ 2)	$\surd$
- context-dependent				$\surd$ 1)					
- n-argument		$\surd$							
ISA relationships:									
- specialization		$\surd$	$\surd$	$\surd$	$\surd$		$\surd$	$\surd$	$\surd$
- disjoint specialization				ext	ext		ext	$\surd$	$\surd$
- partition				ext	ext		ext	emb	emb
- categorization					$\surd$		$\surd$	$\surd$	$\surd$
- generalization				$\surd$	ext		$\surd$	emb	emb

$\surd$: explicit support emb: embedded constraint ext: external constraint
1) fragment concept 2) value-based domains only

Table 1. Comparison of Semantic Data Models

4 Relating the Object Model to Various Modeling Paradigms

In this section we relate our object model to two classical database modeling paradigms, formed by purely value-based and purely object-based data models. With the well-known supplier example we shall try to show that both classical directions can be faithfully simulated by the object model. In consequence, the object model also bears comparison with recent data models which take advantage of both directions by combining their capabilities.

4.1 Complex Object Models including Nested Relations

A relational database schema DB consists of a set of database relation names $R_1,\dots,R_n$, in which each relation R_i is associated with a tuple expression T_i having only atomic data sorts as components. Complex object models [1, 18, 26] extend the relational model by allowing the components of a relation to be arbitrary complex type expressions. Hence, a complex object schema can be seen as a single type DB:tuple(R_1:set(T_1),…,R_n:set(T_n)), and a database instance is a rather complex value of type DB (see Fig. 4). The focus of complex object models is in objects with *internal structure*, i.e., complex objects are directly represented by (printable) complex values. For example, (`SNo`:2324, `Name`:(`Last`:"Miller", `First`:"Tom"), `Cities`:{"Berlin", "New York"}) is a possible object of type `SUPPLIER`. Indeed, the term “complex object models” should be replaced by the term “complex value models” to avoid confusion of concepts. It should be clear that we do not need the object level of our object model to simulate complex object models, because values can be already described by the data level.

```
DB:tuple( SUPPLIER:set(tuple(SNo:int, Name:tuple(Last:string, First:string),
                              Cities:set(string))),
          PART    :set(tuple(PNo:int, Name:string, Price:int)),
          SUPPLY  :set(tuple(SNo:int, PNo:int, Quantity:int)) )
```

Fig. 4. Complex object model schema

Complex object models are of captivating simplicity but they have known drawbacks in respect of the representation of object identity. Since real world entities are modeled by tuple values or possibly key values in those tuples, those values are to be repeated whenever a reference to an entity is needed within another object. This leads to superfluous data in both instances and schema. The redundancy at the schema level adds unnecessary complexity to the types, and creates the potential for error when creating or modifying the database types. The redundancy at the data level complicates the algorithms for update propagation [16]. A solution to this problem is given by the introduction of a further level of indirection, called *object identity* [20], which is the fundamental principle of object-based data models. In our object model object identity is supported by abstract object identifiers.

```
schema DB
  base_objtypes
    SUPPLIER < SNo:int, Name:tuple(Last:string, First:string),
               Cities:set(string) >;
    PART     < PNo:int, Name:string, Price:int >;
    SUPPLY   < SNo:int, PNo:int, Quantity:int >;
```

Fig. 5. Object schema simulating complex objects

In Fig. 5 it is shown how complex object types might be simulated within an object schema. Such a simulation means adding abstract object identity but making no use of it. In the case of complex objects it is not possible for there to be *shallow-equal* objects [6], e.g., two different `SUPPLIER`s with the same values in all components. In principle, this is possible in the object model, because in an object schema instance `SUPPLIER`s are primarily represented by abstract object identifiers, and not by (tuple) values. However, by specifying all attributes as keys, it is possible to obtain the tuple values as an isomorphic copy of abstract object identifiers. Please note that in Fig. 5 only the naive translation is considered. Relational constraints like *key dependencies* (e.g., `SNo` is a key for `SUPPLIER`, `PNo` a key for `PART`, and (`SNo,PNo`) a key for `SUPPLY`), or *inclusion dependencies* (foreign keys, referential integrity) (e.g., `SUPPLY.SNo` → `SUPPLIER` and `SUPPLY.PNo` → `PART`), can serve as the basis for further improvement.

4.2 Pure Object-Based Models

Pure object-based models form the counterpart to the value-based complex object models. In pure object-based models every single data item is considered as an object with identity, and complex objects are modeled by *attribute structure.* FDM [28] is a notable example of this class of models.

```
SNo:    SUPPLIER → SNO                PNo:  PART → PNO
Name:   SUPPLIER → COMPOSED_NAME      Name: PART → PNAME
Cities: SUPPLIER → set(CITY)          Price:PART → PRICE

First:  COMPOSED_NAME → FIRSTNAME     Quantity: SUPPLIER × PART → QUANTITY
Last:   COMPOSED_NAME → LASTNAME      ...
```

Fig. 6. Attribute structure of FDM

An FDM schema (see Fig. 6) is given by a set of object types (upper case items in Fig. 6). Relationships between object types are established via a set of single or multi-valued attributes. As shown by the attribute `Quantity`, the domain of an attribute can be given by the Cartesian product of object types (n-argument attribute). In an FDM schema instance a set of actual objects is associated with every object type o, called the *active domain* of o, and a function f is assigned to every attribute a, so that the domain of f is the active domain of the source of a (the Cartesian product of the active domains if a is an n-argument attribute), and the range of f is contained in the active domain of the target of a (or the finite powerset of the target of a if a is multi-valued). Actual objects may be distinguished either as *printable* such as objects of the types `SNO, CITY, FIRSTNAME`, etc., or as *abstract* such as objects of the types `PART, SUPPLIER` and `COMPOSED_NAME`. For example, the above supplier could be represented by an abstract object $\underline{s}$ of type `SUPPLIER` with `SNo`($\underline{s}$) = 2324, `Name`($\underline{s}$) = $\underline{cn}$, and `Cities`($\underline{s}$) = {"Berlin", "NewYork"} where $\underline{cn}$ denotes an abstract object of type `COMPOSED_NAME`.

Simulating an FDM schema by an object model schema (see Fig. 7) means modeling all object types in FDM as base object types in the object model, because only those types have a time-varying interpretation[3]. Since the object model does not directly support n-argument attributes, tuple domains have to be modeled by explicit object types, e.g., the object type `SUPPLY` has been added. Significantly, in FDM a printable object like 5 or "abc" enjoys the same rights as an abstract object such as $\underline{s}$ representing a `SUPPLIER`. For example, it is possible to have an object type `NAME` with printable names in isolation,

[3] Please note that in Fig. 7 we have used a variant of the object model in which a base object type is allowed to have an empty key group. This is because FDM does not take care of object access.

```
schema DB
  base_objtypes
    SUPPLIER       SNo:SNO, Name:COMPOSED_NAME, Cities:set(CITY);
    PART           PNo:PNO, Name:PARTNAME, Price:PRICE;
    SUPPLY         < S:SUPPLIER, P:PART >  Quantity:QUANTITY;

    SNO            < Sno:int >;
    COMPOSED_NAME  Last:LASTNAME, First:FIRSTNAME;
    CITY           < Name:string >; ...
```

Fig. 7. Object schema variant simulating FDM

i.e., the existence of printable objects does not depend on associated abstract objects. Printable object types in simulation are associated with a single key attribute ranging over a data type, thus having printable values as isomorphic copies of abstract identifiers.

Remark. There have been also approaches to object identity without leaving the framework of complex object models by the use of *printable object identifiers*, thus making the vehicle of object identity visible to the user. The resulting models can be considered to be hybrids of value-based and object-based data models (see [16] on this topic). A model which gives a good example for this is LDM [21].

Besides simplifying object reference, the concept of object identity allows *object sharing*, which avoids redundancy in data and schema, *cyclic object modeling*, which breaks with the strict hierarchical structure of complex object models, *encapsulation of object structure* as in object-oriented programming, and *updates on objects* which also makes the definition of update propagation easier. But as pointed out in [22] or [16], and as it becomes clear in Fig. 7, pure object-based models have severe drawbacks, since they force the user to define a new object type every time a complex value is needed. For example, the object types `COMPOSED_NAME`, `SNO` and `CITIES` are dispensable, as there are no other object types sharing objects with these types. In particular, objects of these types do not belong to entities "in the world". Next we turn to models which take advantage of both complex object models and pure object-based models by combining their modeling capabilities.

4.3 Object-Based Models with Complex Values

A model generalizing complex object and object-based models was presented in [22]. In the model of [22] *classes* and *types*, and *objects* and *values* are differentiated. Objects are members of classes and have an identity, while *values* as members of types are items without identity. The type system induced by the model is actually being used in the database programming language O_2. We draw some parallels to our object model in Table 2.

In [22] a schema is defined as a function σ which associates a type with every class (see Fig. 8). Let $o \in$ OBJTYPE be given in our object model, and let $a_1, \dots a_n \in$ ATTR be all attributes defined on o with $a_i : o \rightarrow s_i$. Then we could view o as a class with $\sigma(o) = \text{tuple}(a_1{:}s_1, \dots, a_n{:}s_n)$. A domain is defined as a function δ which associates a value with every object identifier determining the state of an object base. Let $\underline{o}$ be an object of type o. Then we have $\delta(\underline{o}) = (a_1{:}\mu[\text{ATTR}](a_1)(\underline{o}), \dots, a_n{:}\mu[\text{ATTR}](a_n)(\underline{o}))$. However, there is a difference between [22] and our object model with respect to the interpretation of types. In [22] subtype rules make it possible that the value (a:5,b:7) is associated with the type

Table 2. The model of [22] compared with the object model

notions in [22]	counterpart in object model
class (name) $c \in \mathcal{I}_C$	object type $o \in$ OBJTYPE
object (identifier) $\underline{o} \in \mathcal{I}_O$	$\underline{o} \in \hat{\mu}$[OBJTYPE]
type $t \in \mathcal{S}$	sort expression $s \in$ EXPR(SORT)
value $v \in \mathcal{V}$	$\underline{s} \in \hat{\mu}$[EXPR(SORT]
schema $\sigma : \mathcal{I}_C \rightarrow \mathcal{S}$	object types and attributes
domain $\delta : \mathcal{I}_O \rightarrow \mathcal{V}$	given by μ[ATTR]
interpretation	
of types: compatible values	fixed by μ[EXPR(SORT)](s)
of classes: compatible objects	given by μ[OBJTYPE](o)

```
σ(SUPPLIER)= tuple(SNo:int, Name:tuple(Last:string, First:string),
                   Cities:set(string))
σ(PART)    = tuple(PNo:int, Name:string, Price:int)
σ(SUPPLY)  = tuple(s:SUPPLIER, p:PART, Quantity:int)
```

Fig. 8. Schema according to [22]

`tuple(a:int,b:int)`, but also with the types `tuple(b:int,a:int)` and `tuple(b:int)`. Clearly, in the object model the given value can only be of the first type.

```
schema DB
  base_objtypes
    SUPPLIER < SNo:int >
             Name:tuple(Last:string, First:string), Cities:set(string);
    PART     < PNo:int >  Name:string, Price:int;
    SUPPLY   < S:SUPPLIER, P:PART >  Quantity:int;
```

Fig. 9. Final object schema

In Fig. 9 the final object schema is stated. It is equivalent to the schema of Fig. 8 with the exception that in an object schema we have additional means for logical object identity by the presence of keys. We should note that the final schema is obtained also from the schema of Fig. 5 regarding further relational key dependencies and referential integrity. But clearly, the easiest way to obtain an adequate object schema is to use a high-level semantic data model for conceptual design (see Sect. 2), and then map it to an object schema as a matter of logical design (see Sect. 3).

5 Conclusion

After having translated the various data models into our object model we feel that these data models and the relationships among them can be much better understood. We have considered quite different models from several areas and we have tried to point out the distinctions and the common elements in the different approaches. Perhaps our attempts are a first step towards unifying the concepts and terminology of different approaches. At least it is an attempt to provide a common basis for understanding the various data models.

The main idea of the object model was the simulation of complex object types like tuple, set or relationship types by atomic object types with keys, or in other words, the replacement of internal structure by attribute structure, which unifies data-manipulation. With respect to ER models we emphasize that recent ER extensions in the field of aggregation object types [34] or relationships over relationships [31] are also captured by the object model. Secondly we stress the fact that, in spite of dropping subtypes, our object model still provides a simple framework for modeling different kinds of ISA relationships.

By mapping Entity-Relationship and object-oriented data models to the same approach we feel that we are bridging the gap between Entity-Relationship and object-oriented approaches [4] and pointing out the similarities between both worlds. However, future work must also address the integration of object behavior into the object model, for example by using *events* as shown in [19].

References

1. S. Abiteboul, C. Beeri, M. Gyssens, and D. Van Gucht. An introduction to the completeness of languages for complex objects and nested relations. In Abiteboul et al. [2], pages 117–138.
2. S. Abiteboul, P.C. Fischer, and H.J. Schek, editors. *Nested Relations and Complex Objects in Databases*, Springer LNCS series, Vol. 361, 1989.
3. S. Abiteboul and R. Hull. IFO – A formal semantic database model. *ACM Trans. on Database Systems.*, 12(4):525–565, 1987.
4. C. Beeri. A formal approach to object oriented databases. *Data & Knowledge Engineering*, 5(4):353–382, 1990.
5. M.L. Brodie and D. Ridjanovic. On the design and specification of database transactions. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling – Perspectives from Artificial Intelligence, Databases, and Programming Languages*, pages 277–306, Springer (New York), 1984.
6. J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. In *Proc. 10th ACM Symp. Principles of Database Syst.*, pages 291–299, 1991.
7. S. Ceri, editor. *Methodology and Tools for Database Design.* North-Holland, Amsterdam, 1983.
8. P.P. Chen. The Entity-Relationship model – Towards a unified view of data. *ACM Trans. on Database Systems.*, 1(1):9–36, 1976.
9. R. Elmasri, J. Weeldreyer, and A. Hevner. The category concept: An extension to the entity-relationship model. *Data & Knowledge Engineering*, 1:75–116, 1985.
10. G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual modelling of database applications using an extended ER model. To appear in *Data & Knowledge Engineering*, 1992. Preliminary version: Informatik-Bericht Nr. 90-05, Technische Universität Braunschweig (Germany), 1990.
11. D.H. Fishman et al. IRIS: An object-oriented database management system. *ACM Trans. on Office Information Systems*, 5(1):46–69, 1987.
12. M. Gogolla and U. Hohenstein. Towards a semantic view of an extended entity-relationship model. *ACM Trans. on Database Systems.*, 16(3):369–416, 1991.
13. M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Trans. on Database Systems.*, 6(3):351–386, 1981.
14. R. Herzig and M. Gogolla. On data manipulation in an object model. Internal report, Technische Universität Braunschweig (Germany), February 1992.
15. U. Hohenstein and M. Gogolla. A calculus for an extended entity-relationship model incorporating arbitrary data operations and aggregate functions. In C. Battini, editor, *Proc. 7th Intl. Conf. on Entity-Relationship Approach,* Rome (Italy), pages 129–148. North-Holland, Amsterdam, 1988.

16. R. Hull. Four views of complex objects: A sophisticate's introduction. In Abiteboul et al. [2], pages 87–116.
17. R. Hull and R. King. Semantic database modelling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.
18. R. Hull and C.K. Yap. The Format model: A theory of database organization. *J. ACM*, 31(3):518–537, 1984.
19. R. Jungclaus, G. Saake, and T. Hartmann. Language features for object-oriented conceptual modeling. In T.J. Teorey, editor, *Proc. 10th Intl. Conf. on Entity-Relationship Approach,* San Mateo (California), pages 309–324, 1991.
20. S.N. Koshafian and G.P. Copeland. Object identity. In *Proc. ACM Intl. Conf. on Object Oriented Programming Systems, Languages and Applications,* Portland (Oregon), pages 406–416. ACM, New York, 1986. (Special Issue of SIGPLAN Notices, Vol. 21, No. 11, 1986).
21. G.M. Kuper and M.Y. Vardi. A new approach to database logic. In *Proc. 3rd ACM Symp. Principles of Database Syst.*, pages 86–96, 1984.
22. C. Lécluse and P. Richard. Modeling complex structures in object-oriented databases. In *Proc. 8th ACM Symp. Principles of Database Syst.,* Philadelphia (Pennsylvania), pages 360–368, 1989.
23. P. Lyngbaek and V. Vianu. Mapping a semantic database model to the relational model. In *Proc. ACM SIGMOD Symp. on Management of Data,* San Francisco (California), pages 132–142. ACM, New York, 1987.
24. C. Parent and S. Spaccapietra. Complex objects modeling: An entity-relationship approach. In Abiteboul et al. [2], pages 272–296.
25. J. Peckham and F. Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3):153–189, 1988.
26. H.J. Schek and M. Scholl. An algebra for the relational model with relation-valued attributes. *Inf. Syst.*, 11(2):137–147, 1986.
27. M. Schrefl, A M. Tjoa, and R.R. Wagner. Comparison criteria for semantic data models. In *Proc. Intl. Conf. on Data Engineering,* Los Angeles (California), pages 120–125. ACM, New York, 1984.
28. D. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. on Database Systems.*, 6(1):140–173, 1981.
29. J.M. Smith and D.C.P Smith. Database abstractions: Aggregation and generalization. *ACM Trans. on Database Systems.*, 2(2):105–133, 1977.
30. F.J. Teorey and J.P. Fry. *Design of Database Structures.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
31. B. Thalheim. Extending the entity-relationship model for a high-level, theory-based database design. In J.W. Schmidt and A.A. Stagny, editors, *"Next Generation Information System Technology", Proc. 1st Int. East/West Database Workshop,* Kiev (USSR), 1990, Springer LNCS series, Vol. 504, pages 161–184, 1990.
32. S.D. Urban and L. Delcambre. An analysis of the structural, dynamic, and temporal aspects of semantic data models. In *Proc. Intl. Conf. on Data Engineering,* Los Angeles (California), pages 382–387. ACM, New York, 1986.
33. G.M.A Verheijen and J. Van Bekkum. NIAM: An information analysis method. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Proc. "Information Systems Design Methodologies: A Comparative Review"*, pages 537–590, North-Holland 99, 1982.
34. G. Wei and T.J. Teorey. The ORAC model: A unified view of data abstractions. In T.J. Teorey, editor, *Proc. 10th Intl. Conf. on Entity-Relationship Approach,* San Mateo (California), pages 31–58, 1991.

This article was processed using the LaTeX macro package with LLNCS style