Enhanced Support of Relationship Semantics in Object-Relational Database Systems

Nan Zhang, Theo Härder

Department of Computer Science, University of Kaiserslautern
67653 Kaiserslautern, Germany
E-mail: {zhang | haerder}@informatik.uni-kl.de

Abstract

In face of the complex requirements of advanced applications, extended-relational or object-relational database management systems (ORDBMSs) enjoy now more and more popularity due to their promise to overcome the deficiencies of both object-oriented and relational DBMSs. However, object-relational data models (ORDMs) and their supporting technologies are far from mature. Particularly, while data relationships play a very important role in application domains, they are still not well considered by the new database technology. The main purpose of this work is to outline how accurate relationship semantics can be captured from the real world and supported in the setting of ORDBMSs. We present our prototype ORIENT which facilitates the explicit specification as well as the automatic maintenance of relationship semantics. The prototype is implemented and integrated in an underlying DBMS, so that all our proposals and extensions can be put into practical use.

Keywords: Relationship semantics, object-relational database systems, data modeling.

1 Introduction

The support of complex information systems is a tremendous challenge placing many new requirements on DBMSs. Some of these advanced requirements cannot be met by either purely relational or purely OO technologies alone. Correspondingly, much work has been done or is ongoing to combine these two technologies. The overall goal is to effectively model and manipulate complex data and to use OO technologies for application development without losing the benefits of SQL and all the traditional database services. Therefore, upcoming ORDBMSs [2, 4, 5, 14, 19] attract more and more attention from both the research and the commercial realms.

While the enhanced flexibility and extensibility of ORDBMSs are certainly appreciated, the user is also confronted with increasing complexity. Data modeling, as a fundamental issue of DB-centered application development, deserves reconsideration in the new setting. In particular, data structures and operations are extensively supported by object-relational database (ORDB) technology, especially through user-defined types (UDTs) and user-defined functions (UDFs), whereas data relationships and their inherent characteristics are not well addressed. In addition to value-based symmetric relationship representations through primary/foreign keys (PK/FKs) and referential integrity, object-relational row types and reference types allow more flexible modeling of unidirectional relationships [2]. However, they are only based on simple referencing/dereferencing mechanisms that do not reflect rich semantics. Consequently, a lot of requirements such as refined cardinalities and automatic control of abstraction concepts cannot be modeled in a straightforward way and, hence, must be realized by application logic.

Some proposals or standards have taken relationship semantics and relationship services into account, such as PCTE [7], EXPRESS [6], and OMG Relationship Service [13]. They are, however, not DB-oriented, mostly aiming at special domains, and only rudimentally supported by existing systems. Besides, there have also been some efforts to extend OO models with explicit relationships, such as [1, 8, 12, 18]. These efforts are restricted either to simplified relationships (e. g., binary relationships with a few semantic properties) or to specialized relationships (e. g., aggregation). In the database area, the Entity-Relationship Model [3] includes relationships as a key concept. Further developments have led to a broad spectrum of refinements (e. g., [15]) handling abstraction concepts and several semantic properties such as cardinality. However, these models are rarely supported by existing DBMSs.

Therefore, we believe that the general problem of specifying and maintaining relationship semantics still deserves detailed attention. Most importantly, this work should lead to practical results through integration with promising and widely available data models such as the current ORDMs. With the primary goal to provide enhanced support of relationship semantics on one hand and to exploit the expressiveness and extensibility of ORDBMSs on the other, we are carrying out a project to develop ORIENT (Object-based Relationship Integration ENvironmenT). This prototype facilitates explicit specification as well as automatic control of relationship semantics, thereby hiding implementation details from the user.

We will present this work by addressing issues w.r.t. modeling, specification, implementation, and integration into the underlying DBMS. Sect. 2 gives a brief overview of the basic modeling concepts in ORDMs, investigates their potentials as well as shortcomings in supporting relationships, and then proposes our idea of ORIENT. In Sect. 3, complex relationship semantics is analyzed and summarized. As a result, we develop an SQL-like declarative language OrientSQL for relationship specification and manipulation. Sect. 4 introduces the DBMS-exterior components of ORIENT, while Sect. 5 deals with the DBMS-interior extensions to achieve our goal. Finally, we conclude the paper in Sect. 6 with an outlook on the future work.

2 ORDMs and relationship support

In this section, we introduce the new modeling concepts of ORDMs and discuss what they lack for supporting semantically rich relationships. Then, we will outline the basic idea of ORIENT.

2.1 Object-relational modeling concepts

From the viewpoint of data modeling, ORDMs are very attractive since they offer a broad spectrum of constructs to handle complex data.

- In addition to the basic data types of the relational data model, they provide new built-in types such as large objects (LOBs) that are useful for supporting, e. g., multimedia data types.
- Moreover, the user is allowed to define his own data types (UDTs) tailored for specific applications.
  - Distinct data types, the most basic form of UDTs, facilitate the strong typing mechanism.
  - Abstract data types (ADTs) encapsulate internal structure and type-specific behavior (UDFs).
  - Unnamed row types are used to represent nested structures, whereas named row types are used to define types of rows in tables and thus, make it possible for tables to profit from object properties.
- Different other facilities provide a flexible way to organize complex data structures.
  - Reference types represent inter-object connections in a direct way and make queries through reference paths more compact than through *JOIN*s based on PK/FK pairs.
  - Collection data types allow to comprise zero or more elements of built-in data types, UDTs, or collection types, thus, providing a powerful facility for type construction.
  - Inheritance (single as well as multiple), which enables natural variations among types/tables, is supported through types/table hierarchies.
- All these new concepts can be used wherever basic built-in data types can be used, e. g., as attribute types in UDTs, as parameter types in functions or procedures, and as domain types for table columns.

These conceptual building blocks promise to easily manage a variety of information that would be very difficult to handle using traditional tables (see [22] for detailed modeling examples and evaluation). Therefore, the ORDB technology is generally considered to be more suitable for applications with complex requirements on data modeling and processing. Below we elaborate on how well ORDMs meet the demanding requirements of relationship modeling.

2.2 Relationship support in ORDMs

With the new extensions, ORDMs offer various alternatives to model data relationships, which reflect different levels of semantic expressiveness and convenience.

2.2.1 Basic constructs

Like relational data models, ORDMs represent value-based symmetric relationships using PK/FK pairs. The semantics of this construct can be further enriched through referential integrity and referential actions, which will be discussed in the next subsection.

A more direct and flexible way of modeling structurally linked data in ORDMs is through row objects and object references [2]. Relationships can be described as references to instances of other row types. Queries against such constructs can be issued by means of the referencing operator "." and dereferencing operator "->" as shown in Fig. 1.

```
DDL:   CREATE ROW TYPE program_t (
           p_name CHAR (20),
           include SET (REF (module_t)),
            ......);
       CREATE TABLE program OF program_t;
       CREATE ROW TYPE module_t (
           m_name CHAR (20),
           designer CHAR (30),
            ......);
       CREATE TABLE module OF module_t;

Query: Find all programs that include modules designed by Smith.

DML:   SELECT  *
       FROM  program p
       WHERE  FOR SOME m IN p.include (m->designer='Smith')
```

Fig. 1 SQL3 referencing/dereferencing mechanism

The example is taken from the application scenario to be introduced in Sect. 3.1. The set operator *IN* is employed, since the (1:m)-relationship between *program* and *module* is defined through the multivalued referencing attribute *include*[1].

The referencing/dereferencing mechanism does not, however, exhibit any semantic properties. Complex application-desired semantics must be explicitly modeled in terms of operational constructs and enforced by application logic.

2.2.2 Auxiliary means

Relational systems as well as their object-relational successors define the valid states of data by integrity constraints including column constraints (NOT NULL, unique, check), table constraints (unique, referential, check), domain constraints, and assertions. Among them, NOT NULL, unique constraints (UNIQUE, PRIMARY KEY) and referential integrity constraints with referential actions (CASCADE, SET NULL, etc.) can be used to govern simple semantics between referencing and referenced tables. Since they are tailored to the needs of simple, value-based relationships, they are unsuitable for maintaining richer and more general semantics such as composition and sharability (cf. Sect. 3.2).

ORDBMSs also provide database triggers [21]. However, using pure SQL triggers for our purpose has two disadvantages. Firstly, an SQL-only solution lacks expressiveness for managing complex relationships. Secondly, a trigger is associated to a particular table. As a consequence, the semantics of a given relationship is split in n sets of triggers (with n being the number of relationship participants), revealing rather low conceptual clarity.

1. Note that collection types can be of great use to represent (1:n)- and (m:n)-relationships.

Alternatively, stored procedures can be used to ensure semantic properties. Nevertheless, the stored procedure language, an extended SQL with minor program control logic, is still not expressive enough for our purpose.

Finally, UDFs permit to combine the capabilities of SQL with those of an external programming language. As a result, the expressive power of UDFs is considerably higher than that of pure SQL constructs. However, in contrast to triggers, UDFs do not exhibit active properties. Though it is feasible to embed particular object references with rich semantics in UDFs, this solution is neither intuitive nor adequate concerning the way how such “embedded” object references have to be employed in SQL statements.

2.3 ORIENT

The previous discussion indicates that uniform and concise relationship support is missing in current ORDMs. To remedy this deficiency, we have designed a general framework ORIENT (Object-based Relationship Integration ENvironmenT). The framework extends SQL to OrientSQL that includes separate relationship-oriented statements. Besides declarative specifications, ORIENT also facilitates a graphical user interface to enhance the design convenience. The graphical schema editor OrientDraw helps the user to define Entity/Semantic Relationship (ESR) diagrams, which are built on the basis of the ER model with additional constructs to emphasize and exhibit refined relationship semantics. The ESR diagrams are then transformed to corresponding OrientSQL specifications through the schema translator OrientMap.

Our goal is not only explicit definition but also automatic control of semantics. For this purpose, OrientGen, a precompiler for OrientSQL, produces internal processing constructs as well as metadata for managing the specified relationships.

Most importantly, our enhanced relationship support must not influence the usual way in which data are accessed, queried, and manipulated. Hence, we preserve the SQL3 referencing/dereferencing mechanism as well as DML operations and endow them with new meaning in the presence of internal constructs and complex semantics. This can only be achieved through low-level supports from the underlying DBMS, whose internal processing is therefore extended or overloaded.

Fig. 2 gives an architectural overview of ORIENT. In the following section, we will introduce the formalism of OrientSQL that lays the boundary line of our whole work.

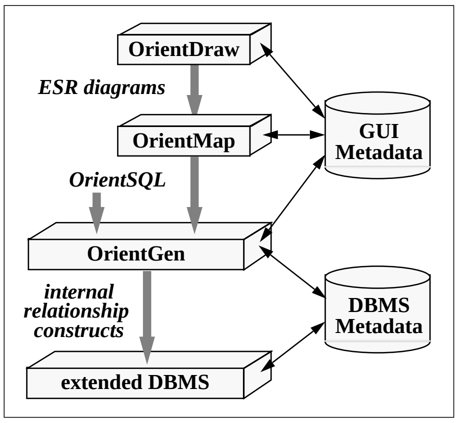


Fig. 2 ORIENT architecture

3 A framework for relationship semantics

As the basis for the OrientSQL definition, the semantic properties of complex relationships will be briefly outlined. Some of these properties have been (partly) analyzed by related work such as [1, 8, 9, 18]. We summarize them to give a full view of our framework for capturing and manipulating relationship semantics.

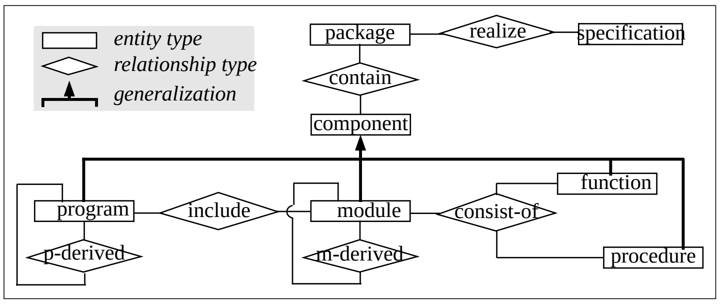


Fig. 3 Application scenario

3.1 Application scenario

We take an example from a software repository environment. An application program consists of modules, a module, in turn, is made up of functions and procedures. All these constituents are represented as components in the repository. All software components are further divided into packages according to the specifications they realize. Programs and modules may be continuously under evolvement. Changes in programs/modules may produce new versions. The dated version might be archived for future reference. The scenario is illustrated in Fig. 3 with ER-like notations.

3.2 Overview of relationship semantics

Complex relationship semantics can be seen as superposition of the following fundamental properties of either structural or operational nature.

3.2.1 Structural properties

- Composition: This property determines how the participants involved in a relationship cohere. In some relationships such as *consist-of*, there is a composition owner, from which operations affecting the whole relationship are propagated to other participants.
- Sharability: This property denotes whether an object can participate in one or more relationships with another object at a higher level. For example, if a function or a procedure belong to only one module, then they are non-sharable in *consist-of*, otherwise, they are sharable.
- Degree: Relationships may be binary, ternary, or of higher degree.
- Cardinality: The cardinality restricts the number of related objects and can be further refined with a pair of values as [min, max], i. e., the minimum and the maximum number of objects that participate in a given relationship instance.

3.2.2 Operational properties

- Existence dependency: Existence dependency characterizes whether an object can exist independently or requires the existence of related objects. To govern this property, certain insertion or deletion operations should automatically incur actions on related objects.
- Transitivity: Besides the operations necessary to ensure existence dependencies, there are (other) database operations that must be executed transitively across complex relationships. For instance, selections at the aggregate level are propagated to the part level. Moreover, this transitivity may be of heterogeneous nature since *UPDATE*, *DELETE*, *INSERT* operations may lead to other operations. As an example, the *m-derived* relationship embodies the semantics that any modification of a module should result in the creation of a new module (version).

Thus, operational properties specify actually consecutive actions that an operation on the participant arouses. Below, we outlines some of the semantics w.r.t. different database operations.

- Deletion propagation
  - **Isolated Deletion (ID):** The deletion of an object has no consequence on other objects.
  - **Mandatory Deletion (MD):** Upon deletion of an object, its dependent objects are also deleted, even though they may be involved in other relationships.
  - **Conditional Deletion (CD):** Upon deletion of an object, its dependent objects are also deleted, if they do not participate in other relationships.
  - **Restricted Deletion (RD):** The deletion of an object is rejected if one of its dependent objects exists. Only all related objects (e. g., an aggregate with its parts) as a whole may be deleted.

- Insertion propagation
  - **Isolated Insertion (II):** The insertion of an object has no consequence on other objects.
  - **Conditional Insertion (CI):** Upon insertion of an object, the relationship is established to existing dependent objects, and absent objects are represented with placeholders (stubs).
  - **Restricted Insertion (RI):** Before the insertion of an object, all dependent objects must be available in the database for establishing the relationship. Otherwise, the insertion is denied.
- Select propagation
  - **Isolated Selection (IS):** The selection of an object returns only this object.
  - **Mandatory Selection (MS):** The selection of an object returns this object and all the associated objects.
  - **Conditional Selection (CS):** The selection of an object returns this object and only those associated objects in the given relationship.

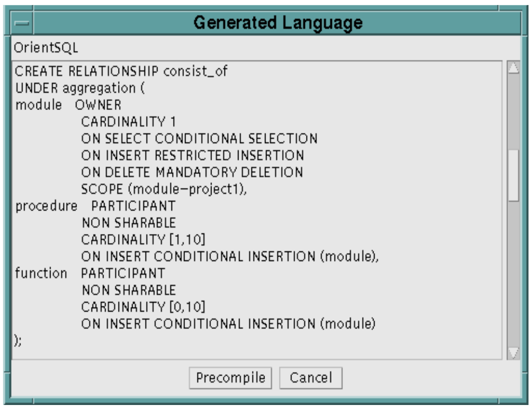
Update as well as heterogeneous propagation semantics are also important in real-world applications and can be refined in a similar way. A more thorough survey can be found in [17].

3.3 OrientSQL

In accordance to SQL, the important aspects of the proposed language are genericity and completeness. Due to space limitation, we can only cover a few examples of OrientSQL. The whole specification syntax (BNF) and explanation can be found in [17].

3.3.1 Relationship definition

New relationship types are defined through *CREATE RELATIONSHIP* statement. Taking *consist-of* in Fig. 3 as example, Fig. 4 shows its definition with specific semantics (generated through OrientMap, cf. Sect. 4.2).



```
CREATE RELATIONSHIP consist_of
UNDER aggregation (
module   OWNER
            CARDINALITY 1
            ON SELECT CONDITIONAL SELECTION
            ON INSERT RESTRICTED INSERTION
            ON DELETE MANDATORY DELETION
            SCOPE (module−project1),
procedure   PARTICIPANT
            NON SHARABLE
            CARDINALITY [1,10]
            ON INSERT CONDITIONAL INSERTION (module),
function    PARTICIPANT
            NON SHARABLE
            CARDINALITY [0,10]
            ON INSERT CONDITIONAL INSERTION (module)
);
```

Fig. 4 Relationship definition

First of all, in our model, relationships can be organized into hierarchies. This can be explained by the observation that on one hand, there are generic, reusable relationship types such as *aggregation* [9], and on the other hand, relationships may be tailored to specific applications. The relationship *consist-of* reflects that a module is composed of functions and procedures. In fact, it is a special kind of the abstraction concept *aggregation* [9] which specifies that the aggregate and all its parts exist as a whole and can be addressed collectively using a high-level query.

By the definition, each relationship type is given a name and possesses a number of participants. Structural and operational semantics is specified w.r.t. each participant. Structurally, *module* plays a special role as composition owner with cardinality [1,1], whereas *procedure* and *function* are non-sharable participants with cardinality [1,10] and [0,10], respectively. As to operational semantics, selection is propagated from the composition owner (*module*) to all participants (**CS**), and the deletion of a module causes the deletion of the procedures and functions included in it (**MD**). A module cannot be constituted without a sufficient number of components (**RI**), which requires the use of the *INSERT BLOCK* statement to build the whole relationship among all participants consistently (cf. Sect. 3.3.2). In contrast, procedures or functions can be inserted even though the module that should consist of them does not exist yet. In this case, the relationship is established with a stub as placeholder for the absent module (**CI**).

Note, in ORDMs, a single row type can be reused to define various tables [22]. For the sake of precise modeling, it is sometimes sensible to restrict a relationship to certain tables. Assume, for example, there are several tables corresponding to the same *module* type, the relationship *consist-of* can be limited to the table *module-project$_1$*. The SCOPE definition is provided for this purpose.

3.3.2 Relationship manipulation

Instead of introducing the manipulation mechanism of OrientSQL, we will only mention several special issues occurring in the new context.

Referencing/dereferencing

The basis of all operations w.r.t. data relationships is the traversal through complex structures. Since OrientSQL enriches SQL3 references with semantics, the referencing/dereferencing mechanism should also be expanded correspondingly. That means, the referencing operator "." should facilitate the traversal path from one participant to another in a given relationship, and the dereferencing operator "->" can resolve the referenced content correctly. We will explain this issue through two simple queries as in Fig. 5.

Query 1: *Find all packages that realize a specification issued by PCTE.*

```
OrientSQL: SELECT *
           FROM package p
           WHERE p.realize -> issuer = 'PCTE'
```

Query 2: *Find all modules that consist of functions written in JAVA.*

```
OrientSQL: SELECT *
           FROM module m
           WHERE m.consist-of (function) -> language = 'JAVA'
```

Fig. 5 OrientSQL referencing/dereferencing mechanism

In the first query, the relationship name (*realize*) to be accessed is used like a referencing attribute in a native SQL3 statement. This is because *realize* is a binary relationship between *package* and *specification*, and thus, will cause no ambiguity by referencing/dereferencing. On the contrary, the second example uses a ternary relationship *consist-of*. To correctly access the attribute *language* in *function* via *consist-of*, the explicit indication of the participant *function* is necessary, since *language* also occurs in *procedure*.

Relationship designation

Since each participant may be involved in more than one relationship and each relationship may possess special selection semantics, it is sometimes necessary to designate a certain relationship (e. g., *realize*) in SELECT. As a result, the selection semantics is restricted in the designated relationship (cf. Fig. 6).

Query: *Find all specifications that are involved in relationship realize and issued by PCTE.*

```
OrientSQL: SELECT *
           FROM specification (realize) s
           WHERE s.issuer = 'PCTE'
```

Fig. 6 Relationship designation

Relationship insert

Except for some special cases in which the initial state of a relationship instance can be automatically established (such as the creation of new versions), the connection of associated participants has to be explicitly issued by the user. To this end, the INSERT INTO RELATIONSHIP statement is provided as some kind of "multi-connect" construct. A new instance can be added to the relationship *consist-of* (cf. Fig. 7) using the following statement, where *m1, f1, f2, f3, p1, p2* are all participant OIDs.

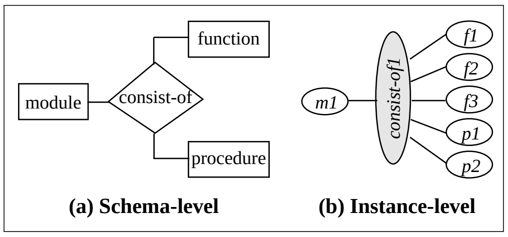


Fig. 7 Schema/instance-level relationship

INSERT INTO RELATIONSHIP consist-of (module, function, procedure) VALUES (m1, {f1,f2,f3}, {p1,p2})

Insert block

In case that all associated participants already exist, the creation of a relationship can be accomplished through a single *INSERT INTO RELATIONSHIP* statement. Instead, multiple native SQL *INSERT* operations for the participants are needed in combination with the *INSERT INTO RELATIONSHIP* statement. According to various insertion semantics (cf. Sect. 3.2.2), a relationship instance can sometimes be constructed without certain participants (according to **Conditional Insertion**), while sometimes it can be added only upon existence of all the participants (e. g., to ensure **Restricted Insertion**). In the latter case, all necessary *INSERT* operations should be executed together. To facilitate the consistency of multi-connections among relationship participants, we provide an explicit *INSERT BLOCK* mechanism.

An insert block is delimited by two statements (*BEGIN INSERTION BLOCK* and *END INSERTION BLOCK*) which group several operations required to build a relationship. Operations defined in an insert block include not only *INSERT INTO RELATIONSHIP* but also native SQL *INSERT* statements. The use of an insert block will delay the checking of the insertion semantics of single operations until the end of the insert block when all relevant operations are executed. Note, an insert block differs from a usual database transaction in that *UPDATE* and *DELETE* operations are not allowed in an insert block, and therefore, if the insertion of any participant violates the relationship semantics, what has to be undone are only insertion operations inside this block. As an example, the insert block defined in Fig. 8 establishes a *consist-of* instance with semantics defined in Fig. 4.

```
BEGIN INSERTION BLOCK    // the beginning of insert block
INSERT INTO function ...    // usual SQL statement
                               // other essential operations
INSERT INTO RELATIONSHIP consist-of
        VALUES (m1, {p1, p2}, {f1, f2})
                               // establishing relationship
END INSERTION BLOCK     // the end of insert block
```

Fig. 8 An insert block

So far, we have presented the framework for relationship specification and for manipulation considering specified semantics. Subsequently, we concentrate on the supporting measures that are implemented partly on top of DBMSs and partly inside DBMSs.

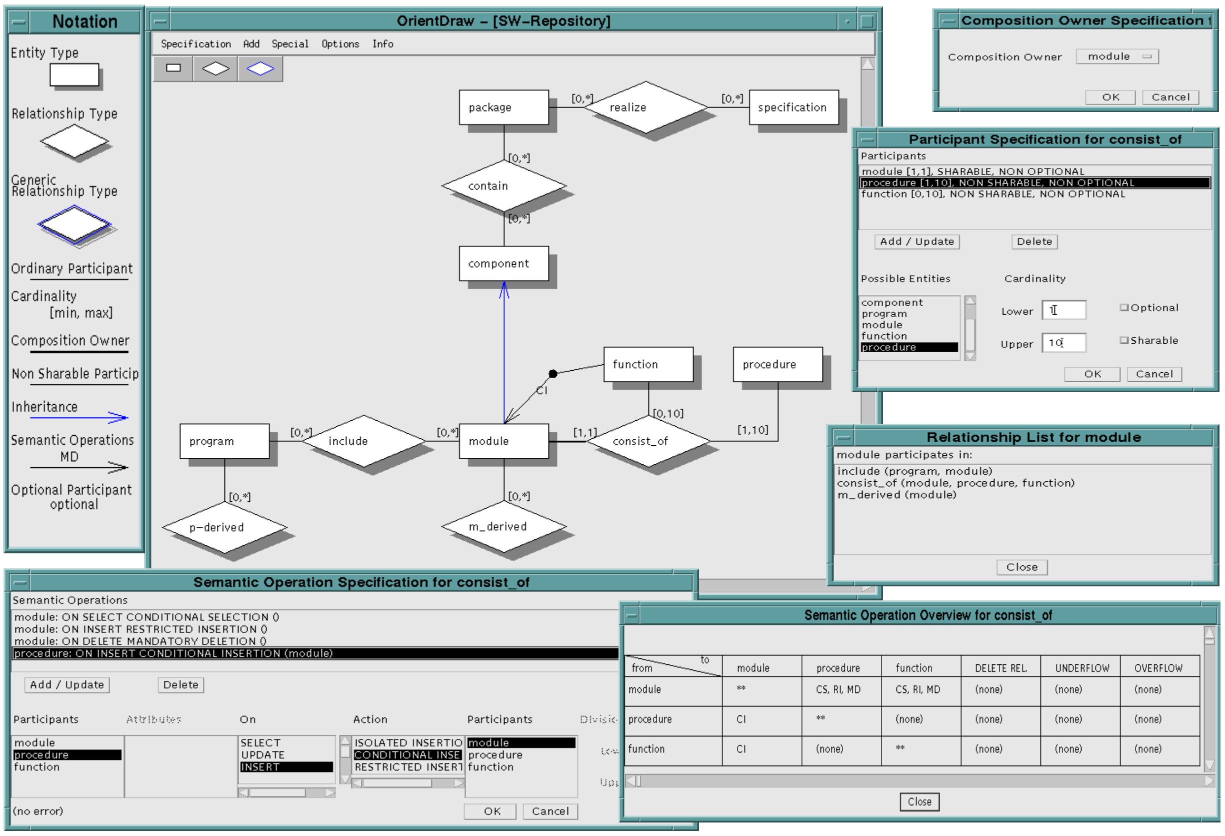
4 DBMS-exterior components

As illustrated in Fig. 2 (cf. Sect. 2.3), the DBMS-exterior components include the graphical schema editor OrientDraw, the schema translator OrientMap, and the precompiler OrientGen.

4.1 OrientDraw

OrientDraw provides an easy-to-use design interface. It keeps the basic ER constructs: rectangles for entity types, diamonds for relationship types, and ovals for attributes. Generalization/specialization relationships are illustrated by bold arrows linking the sub-entities to the super-entity. Different line drawings are used according to various structural properties of relationships (e. g., composition and sharability). Moreover, refined cardinalities can be attached to connection lines. As to operational semantics, each line can be marked with the operation names that need to be propagated from the composition owner to other participants or vice versa. And existence dependencies are indicated through colored lines between two participants and labeled with specific meaning such as **CI** (cf. Fig. 9).

To improve the clarity and readability of the diagram layout, the user may hide some descriptions such as attributes and operational semantics, which can then be displayed in separate diagrams. Besides, our implementation leaves it up to the user to adjust the diagram according to his preference.



Composition Owner: module

Participants
module [1,1], SHARABLE, NON OPTIONAL
procedure [1,10], NON SHARABLE, NON OPTIONAL
function [0,10], NON SHARABLE, NON OPTIONAL

Possible Entities: component, program, module, function, procedure
Cardinality: Lower 1, Upper 10
Optional, Sharable

module participates in:
include (program, module)
consist_of (module, procedure, function)
m_derived (module)

Semantic Operations
module: ON SELECT CONDITIONAL SELECTION ()
module: ON INSERT RESTRICTED INSERTION ()
module: ON DELETE MANDATORY DELETION ()
procedure: ON INSERT CONDITIONAL INSERTION (module)

Participants	Attributes	On	Action	Participants
module		SELECT	ISOLATED INSERTIO	module
procedure		UPDATE	CONDITIONAL INSE	procedure
function		INSERT	RESTRICTED INSERT	function

(no error)

from \ to	module	procedure	function	DELETE REL.	UNDERFLOW	OVERFLOW
module	**	CS, RI, MD	CS, RI, MD	(none)	(none)	(none)
procedure	CI	**	(none)	(none)	(none)	(none)
function	CI	(none)	**	(none)	(none)	(none)

Fig. 9 OrientDraw windows

Standard manipulations are available through pull-down menus. The “Specification” menu contains the usual operations to e. g., open, save, and delete a specification. The “Create” menu offers the possibility to define new entities and new relationships. The “Special” menu provides some special facilities, such as showing the relationship hierarchy, choosing JDBC Driver properties, and calling OrientMap to generate OrientSQL definitions. The “Options” menu contains purely graphical options such as changing the layout. The “Info” menu opens the dialog window to document the information about the user, databases, and schema specifications.

Concrete specifications are supported through context-sensitive pop-up menus, with which various structural and semantic aspects of entities and relationships can be defined and edited.

Fig. 9 illustrates several windows of OrientDraw used to model the example in Sect. 3.1:

- The “OrientDraw” main window display the ESR diagram.
- The “Notation” window exhibits the symbolic conventions of the basic ESR constructs.
- The “Relationship List” window shows all the relationships, in which an entity is involved.
- The “Semantic Operation Overview” window displays all the specified operational semantics of a selected relationship type.
- Several dialog windows such as “Participant Specification” and “Semantic Operation Specification” allow various specifications w.r.t. entities and relationships.

4.2 OrientMap

OrientMap works in two main stages to translate ESR schemas into OrientSQL schemas. In the first stage, SQL3 schemas w.r.t. entity types, inheritance, as well as attributes are generated. With the expressiveness of ORDMs, this mapping process can be well conducted. For example:

- Entity types in an ESR diagram can be mapped into named row types (with corresponding tables).
- Generalization/specialization can be defined as inheritance hierarchies between row types (tables).
- Attributes can be either of built-in types or of UDTs.
- Identifying attributes can be represented using the OID mechanism or primary keys.
- Multivalued attributes can be expressed using collection data types constructors.
- Composite attributes can be described using (unnamed) row types.

In the second stage, OrientMap enriches the schema with OrientSQL statements, so that semantic relationships can be covered. In Fig. 9 we have displayed the design result using OrientDraw. OrientMap transforms this graphical representation into OrientSQL specification partly illustrated in Fig. 4 (cf. Sect. 3.3.1).

In addition to OrientSQL schema, OrientMap also produces metadata that contain information on the ESR schema, the OrientSQL schema, and the mapping of these two schemas. The metadata are essential for supporting later modifications and extensions.

4.3 OrientGen

As precompiler for OrientSQL, OrientGen generates metadata and internal constructs useful for the system to process and manage data according to specified relationship semantics.

The metadata are organized in two levels of system catalogs: The type-level catalogs contain information about relationship types (*relationships*), about the semantics specified between two participants (*references*), as well as the semantics valid for single participants (*participants*). And for each relationship type, separate instance-level catalogs describe the connections and semantics of the relationship instances. We refer readers to [17] for details about ORIENT system catalogs. Below, we will introduce internal relationship constructs produced by OrientGen. Declarative OrientSQL specifications are transformed into internal implementations with a user-transparent process roughly illustrated in Fig. 10.

At first, a separate internal construct (relationship object, ❶ in Fig. 10) is defined as organizational framework for describing, processing, as well as maintaining a relationship. In the relationships with composition semantics such as *consist-of*, objects may play different roles. Moreover, there can be more than one participant in an n-ary relationship. Taking all these into consideration, the relationship object is multi-connected to its associated participants (❷ in Fig. 10).

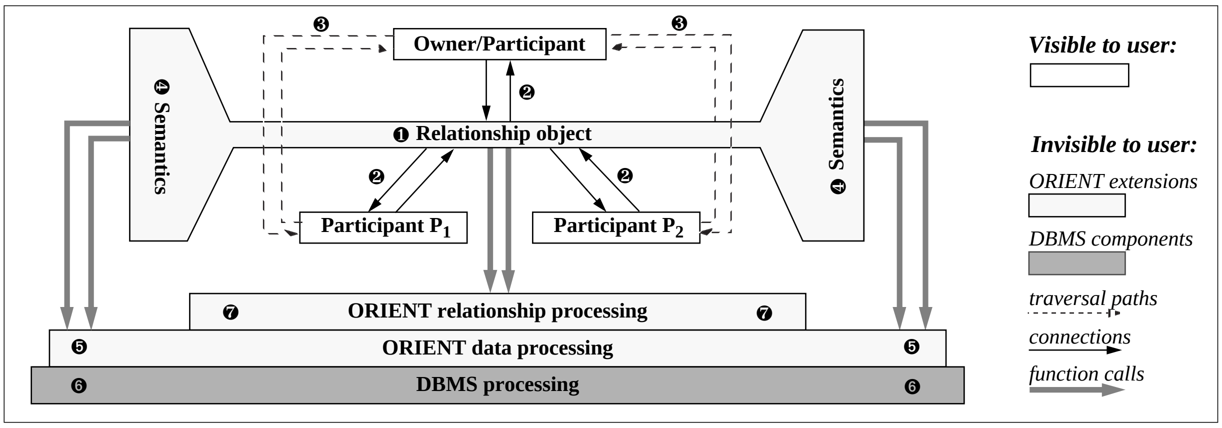


Fig. 10 Internal relationship construct

Then, in the presence of the internal relationship constructs, the logical connections between the participating objects must be resolved through path mediation methods (❸ in Fig. 10), so that it is possible to traverse from one participant to another. This builds the basis for overloading the referencing/dereferencing mechanism.

To maintain specified semantics (❹ in Fig. 10), functional extensions are necessary. The data manipulation operations will be controlled by internal constructs. Here we distinguish between operations issued against relationship participants and operations issued against relationship connections. In the former case, ORIENT's data processing components (❺ in Fig. 10) will take the place of the regular DBMS data processing (❻ in Fig. 10). By doing this, we leave normal DML statements untouched. In the latter case, ORIENT's specific relationship processing (❼ in Fig. 10) will come into play.

Points ❸-❼ in Fig. 10 imply extensions to the DBMS internals and will be further explained in the next section.

5 DBMS-interior components

DBMS-interior extensions are indispensable for two purposes: to let the system automatically control the relationship semantics and to let the user access databases in the usual way.

5.1 OrientSQL processing

SQL statements are processed in relational DBMSs through several steps [10]. In the first step, a syntactic and semantic analysis is carried out. For the statements without search conditions, such as DDL statements, appropriate DBMS internal primitives will be called to handle these statements. For the statements with search conditions, such as DML queries, several other tasks, such as normalization of the transformed queries, are also performed during this phase. As a result, internal representations of the queries are generated for further processing. Subsequently, an algebra graph is constructed for optimization purposes. By replacing logical operators with physical operators, this algebra graph is then rewritten and transformed to an execution plan. Finally, the executable code for the plan is produced with invocations of DBMS processing primitives.

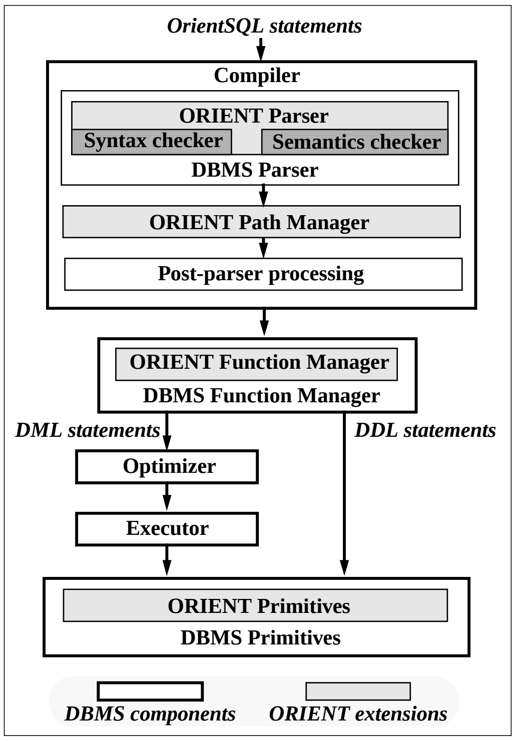


Fig. 11 OrientSQL processing

OrientSQL statements are performed in a similar fashion. Fig. 11 illustrates the steps proceeded by various system components with the ORIENT extensions:

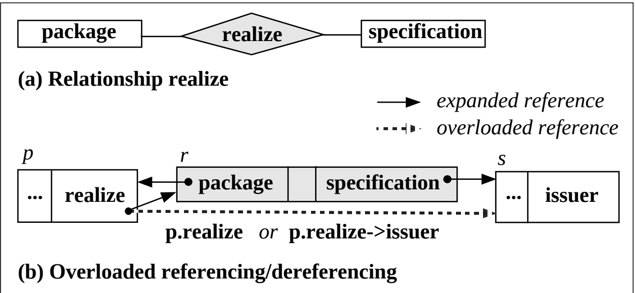
- As an extension of the DBMS parser, ORIENT Parser checks the syntactic and semantic correctness of new OrientSQL statements (e. g., *CREATE RELATIONSHIP*).
- In order to correctly deal with *SELECT* and path expressions in the presence of the new relationship semantics and constructs, the corresponding part of the DBMS parser is overloaded by ORIENT. Moreover, ORIENT Path Manager is responsible for traversals via relationships, i. e., queries with the referencing and dereferencing operators (cf. Sect. 5.2).

- The DBMS component that dispatches the statements with or without search conditions is extended with ORIENT Function Manager to cover new-added statements.
- Most of the ORIENT extensions are related to processing primitives. New primitives are realized from scratch for supporting the new statements such as *INSERT INTO RELATIONSHIP*, whereas existing DBMS primitives that handle *SELECT*, *INSERT*, *DELETE*, and *UPDATE* are overloaded with those taking refined relationship semantics into account. While the algorithms of these processing primitives are described in [17], we will outline below some special problems encountered.

5.2 Overloading referencing/dereferencing

As we have seen in Sect. 3.3.2, OrientSQL lets the SQL3 referencing/dereferencing mechanism continue to work as defined by SQL3. To achieve this, the only reasonable solution is to rewrite the internal operators.

In Fig. 12, referencing from *package* to *specification* and the corresponding dereferencing of *p.specification* where *p* stands for *package* are overloaded to resolve the logical connection between *package* and *specification* through relationship *realize*. An important auxiliary data structure is the *resolution table*. Based on the *resolution table*, the processing of referencing/dereferencing can determine that a referencing attribute (*realize*) from a referencing type (*package*) to a referenced type (*specification*) is augmented with an internal relationship construct (*realize*, names other than that of the original referencing attribute are also allowed). The attribute *referenced scope* is useful in the case that the relationship is restricted to certain tables of the referenced type. With the help of all these, the referencing/dereferencing mechanism can be employed to issue OrientSQL queries like in Fig. 5.



referencing type	referenced type	referenced scope	referencing attribute	relationship construct
package	specification	all	realize	realize
...	...	...	...	...

(c) Resolution table

Fig. 12 Overloading referencing/dereferencing

5.3 Semantics control

Special attention should also be paid to another two issues. The first is that semantics control measures for complex structures should not be taken in isolation. Database operations can affect entire relationship hierarchies. As an example, Fig. 13 illustrates two situations. In Fig. 13a, with the **Conditional Deletion** semantics from *module* to *program*, the deletion of a module (*m1*) causes the deletion of the dependent functions (*f1*, *f2*) and procedures (*r1*) as well as the corresponding relationship.

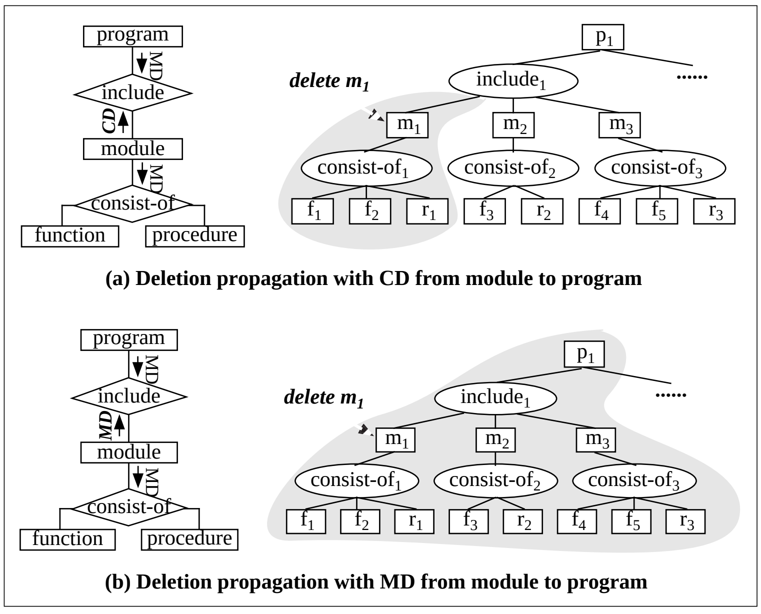


Fig. 13 Operation propagation in complex relationship structure

With the **Mandatory Deletion** semantics, however, the deletion of *m1* will lead to not only the deletion of its own functions and procedures but also the deletion of the program that includes *m1*, which, in turn, results in the deletion of other modules (together with their functions and procedures) included in this program. The whole propagation area is shown in Fig. 13b.

The second is about transaction mechanisms. The realization of relationship semantics requires proper transaction logic, especially in complex structures. Conventional flat transactions are not sufficient in the case of, e. g., implementing insert blocks. Since SQL only guarantees statement atomicity, the most effective way to provide the desired behavior for insert blocks is the use of nested transactions [11]. When the deferred consistency checks do not reveal a violation of the specified semantics, the block concept just behaves like a pair of SQL statements *SET CONSTRAINTS DEFERRED/IMMEDIATE* on a set of related constraints. While in case of a consistency conflict, the subtransaction bracketing the insert block is rolled back thereby providing adequate failure handling semantics.

5.4 Integration with DBMS

In the following, we will elaborate why we have chosen the current implementation platform and which modification or augmentation is performed to this platform.

5.4.1 Platform consideration

Our ultimate goal is to enrich DBMSs with dedicated relationship semantics. To a large extent, the increasing impact of the ORDB technology stems from its extensibility. DataBlades (Informix [5]), DataCartridges (Oracle 8 [14]), or Extenders (IBM DB2 [4]) are the most prominent examples of extended packages provided by many ORDBMS vendors or third-party developers. In contrast to classical extensions for specific application domains by defining a supplementary layer on top of the DBMS core, the object-relational approach results in a closer integration of the new functionality with the existing DBMS features. Such an extension philosophy exactly conforms to the idea of ORIENT — to add features to the user interface of a DBMS by providing new functionality which is integrated into the DBMS core engine.

Closer investigations have shown, however, that the extensibility is still not sufficient in order to provide valuable aid for implementing ORIENT. Generally, the “low-level” extension code is targeted to operations (e. g., input, output, casts, and error handling) on new-defined data types. No change of the DBMS internals is possible.

As we have seen from the previous discussion, to leave the referencing/dereferencing mechanism and the syntax of DML operations (*SELECT*, *INSERT*, etc.) untouched, we have to attack the DBMS core. For these reasons, we decided to take PostgreSQL [16] as the baseline of our prototype. Although PostgreSQL is not a full-fledged product yet, it includes some important OR features such as UDTs, UDFs, inheritance, and SQL. More importantly, as a public domain DBMS, it permits to access and change the source code for achieving a deep integration.

5.5 DBMS extensions

Fig. 14 outlines the DBMS-interior architecture of ORIENT with extensions to various functional components of *Postgres Backend*.

- Parser: *Postgres Parser* is extended with *ORIENT Syntax Checker* (cf. Fig. 11) and *ORIENT Semantics Checker* (cf. Fig. 11), as well as *ORIENT Path Manager* (cf. Fig. 11).
- Traffic Cop: *Traffic Cop* is the main component of *Postgres Backend*. It coordinates the parser,

the optimizer, and the executor. Moreover, system internal primitives are called directly by *Traffic Cop* to process the statements without search conditions ("simple commands"). Therefore, this is the place where *Function Manager* in Fig. 11 plays its role.

- Executor: *Postgres Executor* is extended by ORIENT with two groups of primitives: *Function Primitives* such as *ExecRetrieve (), ExecAppend (), ExecReplace (),* and *ExecDelete ()* are adapted to control relationship semantics during processing data retrieval or modification. *Relationship Primitives* such as *ExecInsertIntoRelationship (), ExecUpdateRelationship (),* and *ExecDeleteFromRelationship ()* are added to handle relationship manipulation.
- Access Methods: *Postgres Access Methods* contain the low-level routines to manage the access of data objects. Also included in this component is *Postgres Transaction System*, which is extended to guarantee the data consistency in the new context.
- Utilities: *Postgres Utilities* consist of different routines, among which *Error Reporting Routines* are most relevant to ORIENT, and thus are considered by our implementation.

Moreover, PostgreSQL is extensible due to its catalog-driven nature. In order to manage ORIENT's metadata that are necessary for relationship handling and semantics control, we have extended *Postgres System Catalogs* with several new tables such as *pg_relationships, pg_references,* and *pg_participants*.

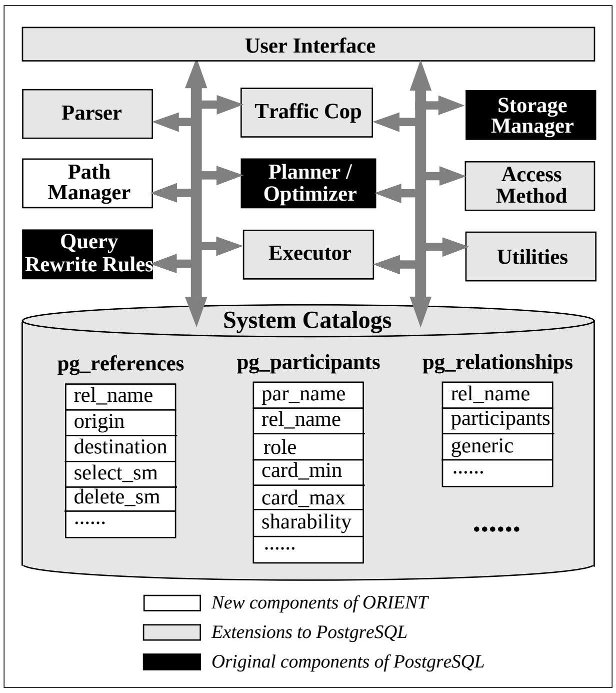


Fig. 14 DBMS-interior architecture

6 Conclusion

The ORDB technology aims at an evolution of the relational data model towards object orientation and, therefore, promises better support for advanced applications as compared to the conventional relational database technology. However, modeling and processing complex data relationships with the object-relational approach remains cumbersome and error-prone. In this work, we have addressed the enrichment of relationship semantics in conformance with the object-relational paradigm of SQL3. We have provided a framework that captures and maintains semantically rich relationships in a straightforward and concise way. To this end, we have introduced not only a set of adequate concepts but also discussed the realization of the prototype, ORIENT, for automatic and transparent semantics control. While the extensions to the DBMS core are realized in C, all the DBMS-exterior components are implemented using Java and communicate with PostgreSQL ([16]) through JDBC (Java Database Connectivity Packet) to store their metadata.

As to the future work, we will continue to complete ORIENT's functionality. To increase the applicability of ORIENT, schema evolution w.r.t. relationship definitions has to be considered. In addition, an appropriate query interface should also be provided in order to achieve our goal, i. e., handling data relationships in a controlled way. We will employ ORIENT in developing real-world applications, thereby demonstrating the feasibility of our approach.

References

[1] A. Albano, G. Ghelli, and R. Orsini: *A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language*, Proc. 17th VLDB Conf., Barcelona, Sept. 1991, 565-575.

[2] ANSI-X3H2-97-315/DBL:BBN-008, *Database Language SQL — Part 2: Foundation (Working Draft)*, Oct. 1997.

[3] C. Batini, S. Ceri, and S. B. Navathe: *Conceptual Database Design: An Entity-Relationship Approach*, Benjamin/Cummings, New York, 1991.

[4] *IBM DB2 Universal Database (Version 5.0)*, IBM Corp., 1997.

[5] *Informix Universal Server (Version 9.12)*, Informix Software, Inc., 1997.

[6] ISO IS 10303-11: *The EXPRESS Language Reference Manual*, 1994.

[7] ISO IS 13719-1: *Portable Common Tool Environment — Abstract Specification*, 1994.

[8] W. Kim, E. Bertino, and J. F. Garza: *Composite Object Revisited*, Proc. 1989 ACM SIGMOD Conf., Portland, 1989, 337-347.

[9] N. M. Mattos: *An Approach to Knowledge Base Management*, LNCS 513, Springer-Verlag, 1991.

[10] B. Mitschang: *Query Processing in Database Systems* (in German), Vieweg Verlag, 1995.

[11] J. E. B. Moss: *Nested Transactions: An Approach to Reliable Distributed Computing*, M.I.T. Press, USA, 1985.

[12] R. Nassif, Y. Qiu, and J. Zhu: *Extending the Object-Oriented Paradigm to Support Relationships and Constraints*, in: Object-Oriented Databases: Analysis, Design and Construction, R. A. Meersman, W. Kent, S. Khosla (eds.), North-Holland, 1991, 305-329.

[13] Object Management Group: CORBAservices: Common Object Services Specification, OMG Document, Revised Edition: March 1995, Updated: Nov. 1997.

[14] *Oracle8 Resource Page*, Oracle Corp., http://www.oracle.com/st/products/uds/oracle8/.

[15] J. Peckham and F. Maryanski: *Semantic Data Model*, ACM Computing Surveys, 20:3, Sept. 1988, 153-189.

[16] *PostgreSQL v6.3.2*, PostgreSQL Org., http://www.postgresql.org.

[17] T. Rieffel: *Semantic Relationships in Object-Relational Database Systems — Concepts, Language Design, and Implementation Approach* (in German), Diploma Thesis, Dept. of Computer Science, University of Kaiserslautern, 1998.

[18] J. Rumbaugh: *Relations as Semantic Constructs in an Object-Oriented Language*, OOPSLA'87, 466-481.

[19] M. Stonebraker: *Object-Relational DBMSs — The Next Great Wave*, Morgan Kaufmann, 1996.

[20] A. Vorberg: *Supporting Semantic Data Relationships — User Interface and Precompiler* (in German), Diploma Thesis, Dept. of Computer Science, University of Kaiserslautern, 1998.

[21] J. Widom and S. Ceri (eds.): *Active Database Systems — Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann, 1996.

[22] N. Zhang and T. Härder: *On Modeling Power of Object-Relational Data Models in Technical Applications*, Proc. 1st East-European Symp. on Advances in Databases and Information Systems (ADBIS'97), St. Petersburg, Sept. 1997, 318-325.