Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Informationssysteme

COCOON and KRISYS

- a survey and comparison -

C. Laasch, C. Rich, H.-J. Schek, M. Scholl,

S. Deßloch, T. Härder, F.-J. Leick, N.M. Mattos

Technical Report No. 195

March 1993

Author's addresses:

C. Laasch, C. Rich, M. Scholl : University of Ulm, CS Department, P.O.Box 4066, 7900 Ulm, Germany, e-mail: {laasch,rich,scholl}@informatik.uni-ulm.de.

H.-J. Schek : ETH Zürich, CS Department, ETH Zentrum, CH-8092 Zürich, Switzerland, e-mail: schek@inf.ethz.ch

S. Deßloch, T. Härder, F.-J. Leick : University of Kaiserslautern, CS Department, P.O.Box 3049, 6750 Kaiserslautern, Germany, e-mail: {dessloch,haerder,leick}@informatik.uni-kl.de

N.M. Mattos: IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose - CA - 95150, USA, e-mail: mattos@stlvm14.vnet.ibm.com

Abstract

The design of Object Based Systems, which map object oriented–, semantic– or knowledge representation models to an underlying database kernel system, in order to close the gap between application and database systems, are currently considered as candidates for future database architectures. KRISYS (University of Kaiserslautern) and COCOON (ETH Zürich), as two prototype systems following this kernel architecture approach, will be compared in this report. We give an overview of the underlying models and query languages as well as their mapping approaches and their processing models, compare the corresponding features, and discuss differences in both systems.

Table of Contents

1. Introduction

COCOON and KRISYS have been designed based on the observation that today's databases supporting simply structured data are quite successful in commercial areas, but future applications need the support for more complex structures, with complex integrity constraints built into the model and with powerful, set–oriented query interfaces. The common idea is to have an object–oriented model (COCOON and KOBRA respectively), supporting flexibility trough powerful structuring primitives, rich semantics, and encapsulation, as well as a powerful query and update language (COOL and KOALA respectively), based on a database kernel (DASDBS and PRIMA, respectively) which efficiently supports common database operations, such as storage of and access to complex database records (see Figure 1.1).

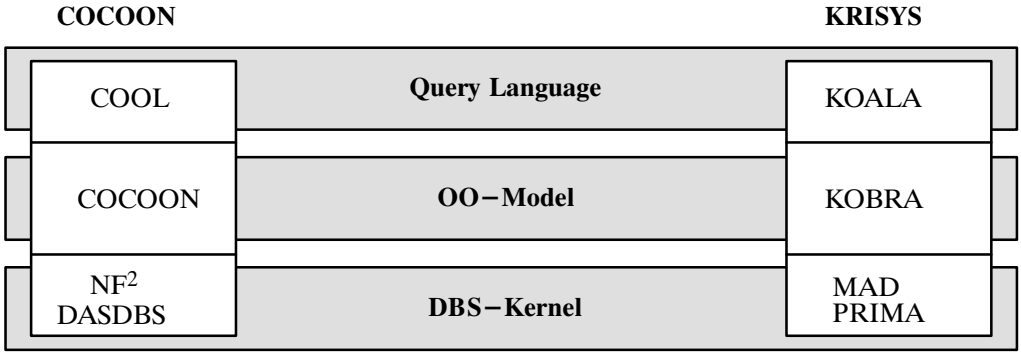


Figure 1.1: COCOON and KRISYS

In order to support the realization of complex applications, it is necessary to provide semantically enriched modeling facilities for representing the objects of the application domain. Both data models, COCOON and KOBRA, offer a variety of concepts to model complex objects. Class and type hierarchies (i.e., classification and generalization), the abstraction concepts of aggregation and association, and methods offer a flexible modelling framework. Additionally, when considering object modeling issues, one can regard not only the efficient object management on behalf of the modeling concepts and their inherent semantics, but also the direct support of the modeling activities (i.e., object base design) by the system as important for application development.

One of the fundamental innovations of the relational approach to databases are the non-procedural query languages, which offers high level, set-oriented database access as well as efficiency through optimizations performed by the DBS as key advantages. Provided that we accept the requirements of new applications to continue to work with such high-level languages in object-oriented DBMSs (OODBMSs), we have to: (i) extend ”relational-style” languages property, so as to adopt them to the more powerful models, and (ii) extend ”relational-style” query processors to efficiently execute these new query languages. Alternatively, one could come-up with new query language paradigms and new processing strategies. It was one of the main objectives of the COCOON and KRISYS projects to reach some (at least preliminary) conclusions on the feasibility and performance trade-offs of these approaches.

It is clearly crucial for the success of OODBMSs to find efficient implementations that improve on the performance of relational systems, rather than being powerful in terms of modelling and features, but just too

slow to be used. Up to now, no final conclusion can be drawn on how the architecture of an OODBMS should look. It is, however, a common anticipation that relational systems as the underlying storage engine would be to slow. Both groups, Kaiserslautern and Zürich had developed prototype database kernel systems that support more flexible structures with operators that go beyond relational queries (PRIMA and DASDBS). They were used as target platforms for the research described below.

The flexibility concerning the modeling and processing inherent in the object models and query languages poses a high demand on the implementation. Even more than in traditional databases, the conceptual structure is difficult to map to an efficient, internal one. Rather, data representing the conceptual objects may be structured completely different for performance reasons. Therefore, the mapping of objects of the models into the structures of the DBS kernel systems, which function as internal interfaces in our systems, has to be considered with great care. The most favorable internal representation is, for example, dependent on the type and frequency of expected queries, that is, the transaction load faced. On the other hand, in order to support efficient application execution, all issues related with query evaluation and optimization as well as general considerations about the mapping of functionality onto the kernel system have to receive special attention. Adequate solutions with respect to these issues have to reflect, among other things, the hardware environment for which the system is conceived.

Even though fairly similar in the overall objectives and even in the general architectural approaches, the two projects differ quite substantial in several aspects. It is the purpose of this report -besides giving an overview of the projects- to provide a synoptical comparison and (rather preliminary) evaluation of the two architectures. More work, in particular practical experiments, has to be done in order to come to more substantial conclusions.

The paper is organized as follows: after introducing a running example in Section 2, we describe the model and operations of COCOON and KRISYS in Sections 3 and 4 respectively, before we compare the corresponding features in Section 5. In Section 6 and 7 we describe the mapping approaches to the underlying kernel systems DASDBS and PRIMA respectively, and in Section 8 we compare these two approaches. Finally we give a conclusion in Section 9.

2. Running Example

In the following we will describe an architectural design application, that we will use as our example throughout this report to explain the models and operations of COCOON and KRISYS, as well as the mapping to the underlying kernels, DASDBS and PRIMA respectively. The description is rather informal, as we just want do describe the scenario we are working with.

In our architectural design application, we are mainly dealing with rooms, furnishings and areas. All of them are design objects. Rooms are described by their name, orientation, position, size, etc. Size, for example, may be further described by the unit aspect, like square meters. Between certain rooms there is a neighborhood-relationship. Further, there are special rooms like parent-bedrooms, which are a subset of bedrooms, which again are a subset of rooms. There are a lot of constraints, such as if a room is

classified as a bedroom or as a parent-bedroom, it has to contain at least one bed or a double-bed respectively, and it has to have a bathroom as a neighbor-room. Each room contains a set of furnishings, which are described by their name, price, width, length, height, etc. Each room belongs to some areas, and each area consist of some rooms. That is, the room 'Office1' could be part of the area 'working-area', and may contain furnishings like 'Wooden Desk' and 'Computer'. Further, we may distinguish furnishings already owned by the user from those which are proposed by the architectural system ...

Example queries in this scenario could be:

- Get the name of all rooms, belonging to area 'working-area', having no 'desk', 'chair' and 'computer'.
- What furnishings are in rooms, that are on the south side of a house?

Figure 2.1 gives a graphical representation of a possible COCOON model for this application. Notice, however, that this graph represents just the core of our example, which will be extended on demand to illustrate special features of the models, COCOON and KRISYS.

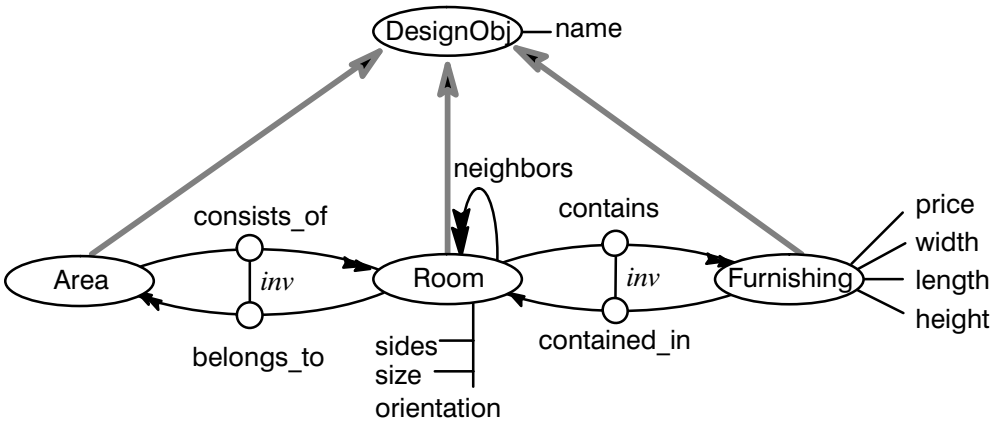


Figure 2.1: Our example DB−World

3. COCOON

Essentially, the COCOON model as described in [SS90a, SS90b] is an object-function model (cf. [WLH90, DMB+87]). Its constituents are objects, functions, types and classes. The query language, COOL, offers object-preserving as well as object-generating generic query operators plus generic update operators. The key objective in the design of COOL was its set-oriented, descriptive characteristics, similar to a relational algebra. In fact, COOL can be seen as an extension of our nested relational query language [SS86].

3.1 Basic Concepts

COCOON is a core object model, meaning that we focus on the essential ingredients necessary to define a set-oriented query language. For instance, tuples as a type constructor are excluded from the core. Basically, all we need are objects (concrete and abstract) and one type constructor, namely set. Other features can be added later due to the orthogonality of the language.

Objects are instances of abstract data types (ADTs). They can be manipulated only by means of their interface, a set of functions.

Data are instances of concrete types (such as numbers, strings) and constructed types (such as sets). The distinction from objects is similar to [Bee89].

Functions are described by a name and signature (i.e., domain and range types). Functions can be single- or set-valued, they are the interface operations of types. The implementation is specified separately (we do not show this here). Notice that we distinguish retrieval functions from methods, that is to say, functions with side-effects. Unless stated otherwise, we use the term functions in the general sense in the sequel. Retrieval functions are a uniform abstraction of ”attributes” and ”relationships” of classical data models. A useful feature is the capability of defining inverses of functions. This integrity constraint is enforced by the system during updates. For an example, see below under ”types”.

Types describe the common interface of all instances of that type, that is, the collection of applicable functions and their interaction. The latter gives the function’s semantics, which can be specified in the form of axioms, for example. We do not consider this any further here. So, the definition of a type basically consists of two parts: a set of functions and a type name. The following example defines a type *RoomT* with six functions *orientation*, *sides*, *size*, *neighbors*, *belongs_to* and *contains*.

***type** RoomT **is_a** Design_ObjectT =*
orientation: string,
*sides: **set_of** Line,*
size: integer,
*neighbors: **set of** Room,*
*belongs_to: **set of** Area **inverse** consists_of,*
*contains: **set_of** Furnishing **inverse** contained_in;*

As we will see later, queries can dynamically produce new types. Those are unnamed, but their set of functions can be derived from the query by standard type inference.

Subtyping. If a type is defined as a subtype of another, e.g.

type *RoomT* **is_a** *Design_ObjectT* = ... ;

then every instance e of the subtype *RoomT*, is also an instance of the supertype *Design_ObjectT*. This is called multiple instantiation. Consequently, all functions defined on the supertype, *Design_ObjectT*, are applicable to the subtype, *RoomT*, too. Further, functions on the subtype may be more restricted than they are on the supertype (that is, their range may be a subtype of that defined in the supertype).

Subtyping defines a partial order "≤" on types. Internally, the type system is completed to form a lattice of types (the powerset lattice of the set of all functions in the database schema), such that for any two types their least upper bound and greatest lower bound are always defined. The top element of the lattice is the most general type *ObjectT* (therefore, all instances of defined types in the database are also instances of *ObjectT*). We allow multiple inheritance, that is, types may have more than one supertype. We assume that naming conflicts have already been resolved (for instance, by prefixing function names with type names).

Classes are strictly distinguished from types in the following sense (see also [ACO85, Bee89]): Types are interface specifications (a collection of functions), whereas classes are containers for objects of some type (type extents). A class *C* is a collection object (an instance of the metatype class). For each class, there is a set of objects, called its extent (*extent(C)*). The elements of that set, called members of the class, are instances of a type, the *member_type(C)*. Due to multiple instantiation, individual members may be instances of several other types too. Even though classes represent polymorphic sets, type checking of our language always refers to the unique member type of the involved sets. A class definition specifies a class name, the member type, the names of superclasses, and an optional class predicate (for the latter two, see below, under subclasses):

class *RoomC : RoomT;*

Due to the separation of types and classes, there may be any number of classes for a particular type (for instance, more than one as the result of selection operations, see below, or none, if we are not interested in maintaining an explicit extent of that type).

Subclasses. There are several choices as to how to define a subclass relationship. Depending on whether the member types of two classes are the same or one is a subtype of the other, and depending on whether the extent of one class is a subset of the extent of the other. That is, we have two known relationships to consider: subtype and subset. As will be seen, these two often correlate, but they need not. Therefore, we will always distinguish carefully which one of them holds. We will speak of a subclass relationship C1 ⊆ C2, iff for the two classes it is true that *member_type(C1)* ≤ *member_type(C2)* and *extent(C1)* ⊆ *extent(C2)*. Usually, at least one of the ordering relationships will be proper. Consider the following example:

type *Design_ObjectT* **is_a** *ObjectT = ...;*
type *RoomT* **is_a** *Design_ObjectT = ...;*
class *Design_ObjectC: Design_ObjectT;*
class *RoomC : RoomT* **some** *Design_ObjectC;*
class B*edroomC : RoomT* **some** r: *RoomC*
where s*elect[select[name="bed"] (contains) O] (r);*

The class *BedroomC* is a subclass of *RoomC* with the same type, but a subset of the objects, whereas *RoomC* is a subclass associated with a subtype of *Design_ObjectT*. The predicate given for class *BedroomC* is a constraint that all members of *BedroomC* have to contain a bed. This is a necessary, but not sufficient condition for members of *RoomC* to become members of *BedroomC*. Changing the keyword

some to **all** would indicate a necessary and sufficient condition: in this case the DBMS would automatically classify rooms into the subclass, if the predicate evaluates to true. Notice that, unlike e.g., we define the extent of a class to include the members of all its subclasses.

Views are derived classes, where the member type and the extent is defined implicitly by a query expression. Thus, the extent of a view is usually not stored explicitly, but rather computed from the view-defining query. Views provide a specialized interface to some base objects. A user or an application programmer usually works only with a small portion of the global schema, a *subschema*, that is particularly tailored for the task performed. Such a subschema consists of a collection of base and/or view classes together with the functions defined on them.

Our view mechanism allows arbitrary queries to serve as view definitions, exactly as in relational DBMSs. For example:

define view *LargeRoomV* **as project** [*name, neighbors, size*](**select** [*size* > 16](*RoomC*)

The view *LargeRoomV* contains the (sub)set of rooms whose size is bigger than 16. According to the projection only the functions *name, neighbors,* and *size* are applicable to members of this view. The other ones, like *consists_of*, and *contains* are hidden from the user.

Views can be used as arguments for queries and updates, just as ordinary classes can. We will see that, in contrast to the relational model, only a few restrictions have to be imposed. In fact, all update operators have the same effect as if they were applied to the base class, because the views' extents are derived from them.

Variables. Due to the set–oriented style of the query language, objects are typically unnamed. However, variables can be used as temporary names ("handles") for objects. They have to be declared with their type, such that compile–time type checking applies to variables too. For example,

var *My_Room : RoomT*;

declares a variable *My_Room* of type *RoomT* that can, for example, be assigned the result of an object creation in order to identify the new object in subsequent (update) operations.

3.2 Object Algebra

The key objectives in the design of COOL has been its set-oriented, descriptive characteristics, similar to a relational algebra. In fact, COOL can be seen as an extension of our nested relational query language [SS86]. The main characteristics of the COOL query algebra are the three following: First, it is a strongly-typed algebra that allows static type-checking. This is achieved by using the type associated with a class to check whether the operations on the class members are legal. Secondly, the COOL operations are set-oriented, where the inputs and outputs of the operations are sets of objects. Hence, query operators can be applied to extents of classes, set-valued function results, query results, or set variables. These are collectively called <*set-expr*> in the sequel. Formally, a class name *C* as an argument is a shorthand for *extent(C)*. Thirdly, the operations have object-preserving semantics, such that the results of queries are (some of) the existing objects from the database.

Selections (**select** [*P*] (<*set-expr*>))**.** A selection returns a subset of the input set of objects, namely those satisfying the predicate *P*. The type of the set is unchanged, i.e., it is type(<*set-expr*>).

Projection (**project** [*A1, ..., Ai*] (<*set-expr*>)). The output of a projection is a set with a usually new type, a supertype of the input type, as less functions are defined on the output, namely only those listed in the projection. All objects of the input set are also elements of the output set (*object preservation*).

Extend (**extend** [<*fname*>*:=*<*expr*>*,...*] (<*set-expr*>)). Projection eliminates functions, extend defines new derived ones. Obviously, each given function name <*fname*> must be different from all existing functions for the type of the input. <*expr*> can be any legal arithmetic-, boolean-, or set-expression. The objects of the input set are preserved, that is, extend returns a set with the same objects as the input, but with a new type, a *subtype* of the input type (all the old functions plus the new ones are defined on it).

Set Operations. As the extent of classes are sets of objects, we can perform set operations as usual. One notable point is the criterion for duplicate elimination (or equality determination): equality of abstract objects is what this usually called identity. So, this is the notion that is used in the set operations. With a polymorphic type system, we need no restrictions on operand types of set operations (ultimately, they are all objects). The result type, however, depends on the input types: for the operation **union** it is the lowest common supertype (in the lattice) of the input types. The **difference** operation yields a subset of its first argument with the same type; finally, **intersection** results in the greatest common subtype.

Pick (**pick** (*set-expr*)) is provided to convert a singleton set into the element (i.e., drop the spurious braces). The result of applying **pick** to a set with more than one element is undetermined in the sense that one randomly selected element is returned.

These are the basic *object preserving* query operators of our algebra. Other operators, such as join can be derived from them. The complete algebra includes an operator (**extract**) for generating sets of *tuples* as query results to communicate with value-oriented environments. Formally, we do not provide object generating *query* operators. However, new objects can be derived by combining a query with an update operator, **create**, that generates the new objects.

3.3 Update Operations

In OODBMSs updates are usually carried out by type-specific methods that are defined by the type implementor. These methods are means to offer integrity-preserving updates to *clients*, but there is no support for method *implementor* as far as integrity preservation is concerned. Therefore, COOL provides generic update operations that maintain model-inherent integrity constraints, and make the implementation of methods simpler. Additionally, applications might make direct use of these update operations. Therefore, object manipulations that involve the generic modeling facilities of COCOON, do not necessarily need to be coded into methods over and over again. Rather, applications can use our generic operators directly where appropriate.

The update operators can be divided into four groups, from which the first three groups are defined according to the three modeling concepts variables (including functions), types, and classes: Assignments,

operations for object evolution, operations for manipulating the extents of classes and views, and type-specific methods. The last group are sequences of update operations that allow to keep complex integrity constraints consistent. All update operations are applied to single objects instead of sets. However, in order to allow for set-oriented updates we provide a descriptive iterator (**apply_to_all** [*upd-seq*](*set-expr*)) that takes a sequence of update operations *upd-seq* as a parameter that is executed for each element of a set *set-expr* (e.g., a query result). Thus, we can not only apply any single generic update operations in a set-oriented fashion, but also update sequences or type-specific methods[LS93].

3.3.1 Assignments

Assignments to Variables (*v* := *e*).The value of a variable *v* can be explicitly modified by an assignment. As usual for object-oriented languages, the inferred type of the right-hand side expression *e* can be a subtype of the variable's type.

Since functions are also regarded as variables, it is possible to assign sets of pairs that are produced by appropriate expressions to functions. The effect would be that the function is redefined for all arguments at the same time. Typically, however, we want to redefine function values only for particular arguments. This is achieved by the following.

Partial Assignments to Functions (**set** [*f* := *e'*](*e*)). The operation set changes the function *f* such that the function application *f(e)* results in the value of the expression *e'*.

Notice, however, that objects that are denoted by variables might change their function values, although the variables are not used. This is due to object sharing, one of the most important properties of object-oriented models (see also object-oriented programming languages such as Eiffel [Mey88]). The ''usual'' semantics of such an assignment in databases is the snapshot semantics, that the query is evaluated once and the result is assigned to the variable. This distinguishes set-variables from database views, which are reevaluated each time the view is used. However, in object-oriented models the snapshot semantics of assignments interacts in a subtle way with object sharing.

3.3.2 Operations for Object Evolution

Object Creation (**create** [*T*](*v*)). The creation of an object by **create** [*T*](*v*) instantiates type *T* and assigns the new object to the variable *v* (whose type has to be *T* or a supertype thereof). Notice that object creation involves ''invention'' of new OIDs, that is, the object (OID) assigned to *v* has to be different from all existing ones. In general, **create** changes the database state such that the active domains of all types *T'* that are supertypes of *T* (and *T* itself) are extended by the newly created object. Thus, by making the created object an instance of *T* and all its supertypes, the subset semantics of the subtype relationship is maintained.

Dynamically Acquiring More Types (**gain**[*T*](*e*)). The **gain** operation is used to dynamically establish new instance-type relationships between an object denoted by *e* and a type *T*. It does not change any values of variables or functions, but it adds the object denoted by *e* to the active domains of type *T* and all its supertypes.

A possible extension of the semantics could be to add the specification of default values for functions that now become applicable. Currently, none of the new functions gets a value, that is, they are all undefined for the object *e*.

Dynamically Losing Types (**lose**[*T*](*e*)). In contrast to the **gain** operation, **lose** deletes instance-type relationships. This is, the object denoted by *e* is removed from the active domains of *T* and all its subtypes. The effect of the operation is that all functions that are defined on the type *T* or a subtype of *T* are no longer applicable to the object denoted by *e*. Additionally, since COOL is a strongly-typed language and objects might be shared, the semantics of the **lose** operation has to be defined with respect to all typing constraints of variables and functions. As a consequence, we have to remove each occurrence of the object denoted by *e* from variables, sets and functions, if they typed with *T* or a subtype of *T*.

This proposed semantics guarantees that static type checking is sufficient despite operations that change types of objects dynamically. More intuitively, this kind of semantics implements the point of view that type information is part of the constraints that every valid database state has to fulfill. Once such constraints fail to hold, the state is changed by removing the objects from variable values.

Deletion. We need no explicit **delete** operation, since its functionality is subsumed by the **lose** operation, if an object loses its most general type *ObjectT.*

3.3.3 Manipulating the Extents of Classes and Views.

The operation **add**[*e*](*C*) is provided to add an already existing object denoted by *e* into the extent of the class *C*. According to the subset–semantics of the subclass relation, the object is also added to the extents of *C*'s superclasses.

An object denoted by *e* can be removed from the extent of the class *C* by the operation **remove** [*e*](*C*). This operation has no effect on the existence of the object, but only on the membership relations between the object and classes. This is, the object is not only removed from the extent of the class *C*, but also of the extents of *C*'s subclasses, since these extents have to be subsets of *C*'s extent.

A detailed discussion of the semantics of the operations **add** and **remove** according to class predicates and views can be found in [LS92].

4. KRISYS

4.1 The KOBRA Model

An OODBMS should provide not only efficient and reliable management of object bases (OB) but also means for constructing such OB in a step by step fashion. It should be flexibly used as a tool for dynamically defining OB contents during application development and efficiently employed for OB manipulation during application processing.

Considering an appropriate framework for OB construction, the following modeling features have been observed as especially important in the KRISYS project:

- ***Means for supporting an incremental application development:*** During the construction process, the OB-designer should use the object model as a design tool. As he/she extends and modifies the OB schema or contents, the system should dynamically reflect the consequences of design modifications and control the consistency of the design, preserving any kind of information already specified even in the case of complex redesigns or reformulations.
- ***Elimination of the difference between OB schema and OB contents:*** The model should explicitly integrate meta-information (e.g., descriptions of object types or classes) into the OB, eliminating any difference between OB schema and OB contents. A strict separation, as required by conventional DBMS, neither enables an incremental OB construction nor reflects real world situations, where changes affecting the application model may also occur (e.g., in the case of schema evolution).

In order to support the above requirements, the model of KRISYS exploits an integrated view of abstraction concepts [Ma88a]. Such abstraction mechanisms, in contrast to existing DB techniques, advocate a modeling process in a stepwise fashion [BMW84], enabling an integration of meta-information into the OB as well as an incremental specification and refomulation of the OB contents. In the following, we discuss such exploitation of abstraction mechanisms for application development, thereby providing a brief look at the object model of KRISYS. For this purpose, we will only introduce the concepts necessary for a better understanding of our considerations, instead of providing a complete description of the KOBRA model (see [Kr89,MM89] for details).

4.1.1 An Overview of the KOBRA model

The KOBRA model [Kr89,MM89] integrates descriptive, organizational, and operational aspects of real world objects into one basic concept, called *schema* (not to be confused with a DB-schema), which is used to represent every entity of the modeled world (e.g., 'my-living-room' in figure 4.1). A schema (others call it frame, unit, or simply object) is uniquely identified by a name (i.e., object-identifier), and contains a set of attributes to describe its characteristics. Attributes can be of two kinds: *slots* are used for the representation of properties of a schema (e.g., 'size', 'orientation') and of its relationships to other schemas (e.g., 'neighbors' referring to adjacent rooms); *methods* are used for expressing behavioral aspects of this entity (e.g., 'make-two-rooms', which is used when designing the floorplan of a house to consistently divide a room into two separate rooms). In order to characterize an object in more detail, all kinds of attributes can be further described by *aspects* (possible-values and cardinality specification, default-value, user-defined aspects, etc.).

For object structuring, KOBRA supports the abstraction concepts of classification, generalization, association, and aggregation. Their semantics are guaranteed by *built-in reasoning facilities* provided by the system and used as the basis for drawing particular conclusions about objects [Ma88a,RHMD87]. These concepts are incorporated into the model by means of special, system-controlled attributes (and as such can also be further specified by aspects). That is, they are seen as special, predefined relation-

ships between objects, defining the overall organization of a OB as a kind of complex network of objects. Hence, each schema can be related to other objects by means of any abstraction concept. Thus, classification/generalization as well as association and aggregation form a directed acyclic graph (sometimes called lattice) rooted in a system-defined schema. Since each schema can be a node in each of these graphs, the OB can be seen as the superposition of three graphs. The same object can therefore represent a class with respect to one object and a set or even an instance with respect to another. 'My-living-room'', for example, represents an instance of room, an element of large-rooms, and an aggregate composed of a sofa, table, and a rocking-chair. Thus, KOBRA supports an *integrated view of OB objects*, i.e., since there are no separate representations for sets, classes, instances, or complex objects, the difference between data and meta-data, which is usually apparent in existing data models, is eliminated in KOBRA so that meta-information is integrated into the OB [MM89].

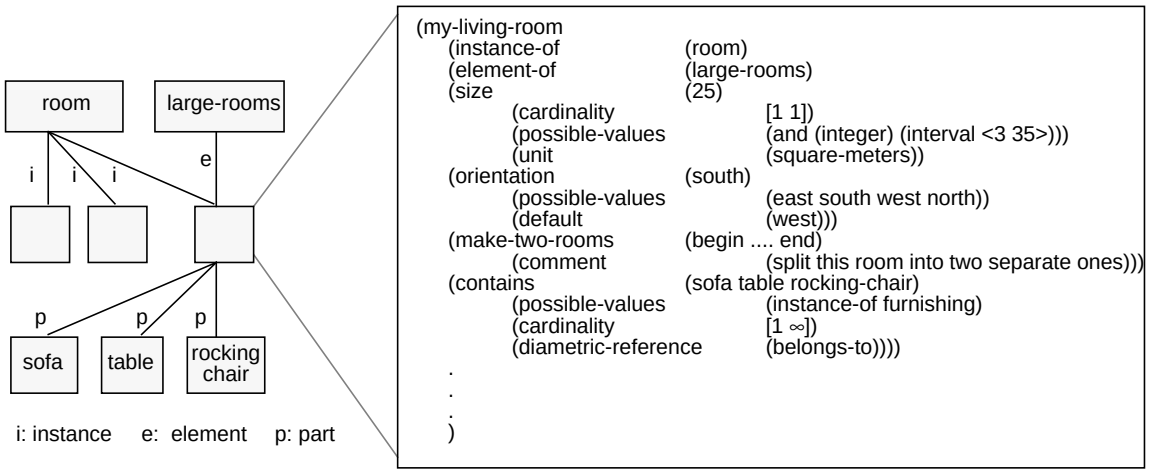


Figure 4.1: Example description of the schema 'my-living-room'

As already mentioned, behavioral aspects of an entity are represented by means of methods. They express operations, which may change the object's properties or relationships, defining a kind of interface to other objects. KOBRA provides two further concepts for the specification of operational knowledge: *demons and general reasoning facilities.* Demons (similar to triggers) are procedures attached to attributes, which are automatically activated when attributes are manipulated or accessed. The general reasoning facility is provided by *rules* which can be evaluated for forward or backward reasoning. Due to space limitations of this paper, it is not possible to present the aspects of rules and demons (for details, see [Ma91]). In the context of our discussion, it is however important to mention that both mechanisms are represented by means of the same concepts described above.

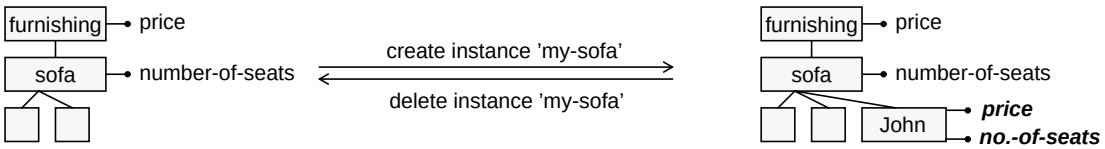
4.1.2 Exploiting KOBRA as Modeling Tool

As already mentioned, special 'automatic' reasoning facilities are supported by KRISYS as part of the abstraction concepts and activated by modification and retrieval operations whenever necessary. The modeler takes advantage of such reasoning facilities [Ma88a,RHMD87] in order to facilitate his task by exploiting the system to make deductions about objects as well as to guarantee the structural and semantic integrity of the OB.

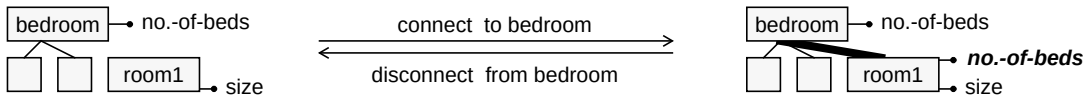
Classification/Generalization reasoning

The most usual of these reasoning facilities is the one built into the is-a hierarchy, i.e., inheritance. Since classes define the structure (i.e., attributes and constraints) of their instances and subclasses, by inserting objects into the OB and expressing the belief of the modeler that they are instances or subclasses of some classes, KOBRA can reason that they have the properties defined by these classes (see figure 4.2a).

However, objects may exist in a KRISYS OB *without the specification of a class*. Frequently, the OB designer knows about the existence of an object but not of its type so that it has to be first introduced in the OB without any instance-of specification. After this, the OB designer might reflect upon its type or he might want to see how the object would look if it were an instance of a particular class (e.g., he 'believes' at first that room1 is a bedroom) and defines an instance-of relationship. (Note that in this case the modeler is using the OODBMS as a tool to determine the structure of the objects, by dynamically defining instance-of relationships.) KOBRA will then deduce the object structure, generating a more detailed description of this object. Based on this new description, the OB designer might now realize that it does not correspond to the real world entity, starting the determination of the object type once again (i.e., he will realize that room1 is not a bedroom, as illustrated in figure 4.2b. Therefore, instance-of relationships correspond, in truth, to current beliefs of the modeler, which may change at any time without affecting the existence of other objects or further descriptions (i.e., properties, constraints, etc.) that an object may possess by its own. (Obviously, the same holds for objects connected by subclass-of relationships).In analogy to a dynamic definition of relationships, attributes or integrity constraints (by means of aspects) can also be defined in a stepwise fashion. In this case, inheritance is employed to ensure that each instance or subclass has, at least, the attributes of its superclasses. If the attribute price is added to the object furnishing, it will be immediately inherited by all its subclasses and instances; if it is deleted, KRISYS automatically removes it from the corresponding subclasses and instances.)



a) introducing / removing instances or subclasses



b) connecting / disconnecting instances or subclasses

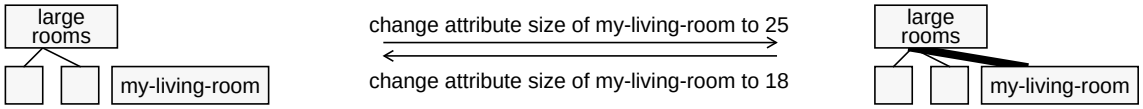
Figure 4.2: Reasoning facilities of generalization and classification

Association reasoning

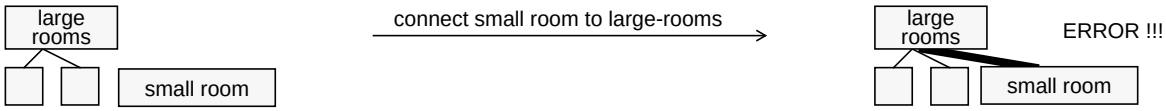
Reasoning on association hierarchies is at first provided by the so-called membership stipulations, i.e., properties that an object must satisfy in order to be added to the group of elements of a set. Since KRISYS guarantees that every element of a set always fulfils the corresponding membership stipulation, it

is possible to deduce which are the elements of large-rooms. Consequently, by specifying membership stipulations based on the structure of the elements (e.g., elements of the set large-rooms must have a value of the slot size, which is greater than 20), KOBRA can determine whether a change in an element should cause the dissolution of an existing association relationship or the creation of a new one because of the satisfaction of the membership stipulation of another set. As such, when one decreases the value of the slot size of a room below 20, this room is automatically removed from the set large-rooms (figure 4.3a).

Additionally, membership stipulations can be used as integrity constraints. If, for example, an attempt is made to explicitly install a room with a size smaller than 20 as an element of large-rooms, thereby violating the condition specified in the membership stipulation, KRISYS will cause an error and prevent the connection (see figure 4.3b).



a) modifying values of attibutes of the elements



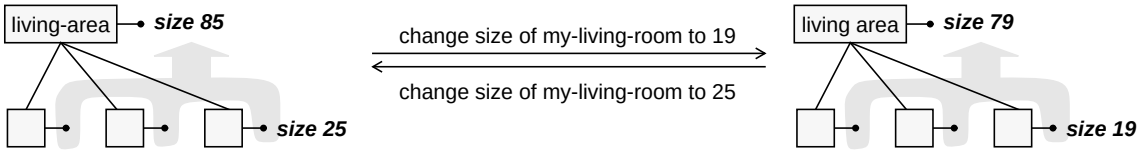
b) connecting explicitly objects to sets

Figure 4.3: Reasoning facilities of association

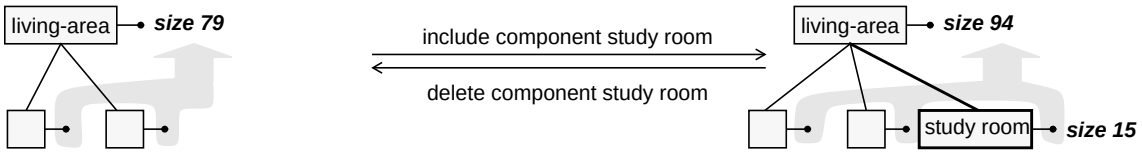
Further reasoning capabilities are provided by so-called set properties, i.e., properties that describe the characteristics of the group of elements as a whole. Large-rooms, for example, has properties defining the average of size of its elements, their number, etc. Since such set properties are in reality based on characteristics of each individual element, conclusions about the values of set properties can be drawn from the elements' attributes. So, upon changes on elements (e.g., because of a modify operation) or on the relationships between elements and sets (e.g., because of the introduction or deletion of an element), the recalculation of the values of set properties is also performed by KRISYS.

Aggregation reasoning

In the aggregation, reasoning is performed based on the specification of the so-called implied predicates, i.e., predicates expressing monotonically increasing or decreasing properties on the aggregation hierarchy. For example, the size of a design-object increases moving upwards on an aggregation (i.e., from furnishings to rooms and areas), since it is dependent on the sum of the corresponding parts' sizes. Hence, conclusions can be drawn by simply guaranteeing the truth of such implied predicates. E.g., when the size of a room is increased, KRISYS automatically reconsiders the sizes of all areas containing this room (e.g., the living-area in figure 4.4a). In the same manner, similar conclusions are drawn, for example, when objects are introduced in or removed from an aggregation, as illustrated in figure 4.4b.



a) modifying values of attributes of components



b) introducing / removing components

Figure 4.4: Reasoning facilities of aggregation

In analogy to the membership stipulations, implied predicates may also be used as integrity constraints. For example, the size of a room can be seen as a constraint for the sizes of its parts (i.e., furnishings). When new furnishings are added as parts of the room or the size of either the room or one of its parts is changed, KRISYS will check the correctness of the implied predicate and report an error on the violation of the predicate.

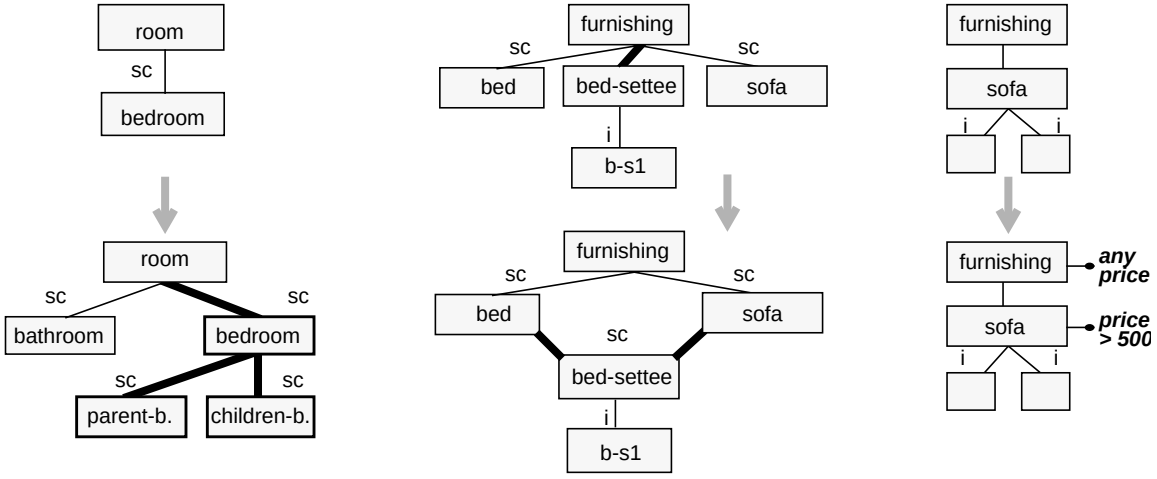
Knowledge reformulations

It is important to observe that the reasoning facilities described above can also be viewed as a means for undertaking the maintenance of the structural and semantic integrity of the OB [De90]. For example, inheritance is employed to ensure that each instance or subclass has, at least, the attributes and satisfies the constraints prescribed by its superclasses. Since KRISYS guarantees that such integrity holds even if changes to the OB structure occur, knowledge reformulations may be carried out effortlessly without affecting any other aspect of the existing knowledge. This feature is essential for application development support since most prototypes perform poorly at the start because of an erroneous transformation of the application world to objects of the object model. For this reason, mechanisms for diagnosing the weakness and deficiencies of the OB are very important in this environment. To support such kind of OB evaluation, KRISYS provides several means, such as for example, simulation of rule-based inference processes, error handling inside of methods, stepwise rollback of the execution of demons, rules, and methods, trace facilities, definition of break conditions, etc. [Kr89].

Refinements or changes of existing specifications are certainly the easiest modifications to be performed on a KRISYS OB. These are directly achieved by using KRISYS operations to define new aspects, membership stipulations, and implied predicates, delete old ones, restrict or generalize them, or even move them up and downwards on the corresponding hierarchies. In the example shown in figure 4.5c, a constraint for the instance-of relationship was introduced and correspondingly specialized to control the insertion of instances of furnishing into the right subclass according to their prices.

Restructuring usually generates new concepts and modifications in the abstraction hierarchies. For example, the application designer is not satisfied with the defined structure of furnishings. He criticizes, that

the fact that bed-settees share properties of both beds and sofas is not represented in the OB. Thus, the OB-designer has to introduce bed-settee as a subclass of bed and sofa (figure 4.5b). The same occurs in the case of redesigns after a new conceptualization of the real world. In order to represent a more extensive classification of rooms, it may be necessary to introduce further subclasses of room and the corresponding instances into the OB (figure 4.5a).



a) redesigns b) restructurings c) refinements

Figure 4.5: Examples of knowledge reformulations

Restructuring and redesigns demand adding new objects and attributes to the OB, inserting new classes, sets, or aggregates into existing hierarchies, moving objects and attributes up, down, or sidewards on such hierarchies, renaming objects and so on. In general, the most radical changes come from adding or deleting abstraction relationships between objects. But even in the case of these radical changes, KOBRA provides direct and flexible ways to achieve such reformulations, keeping the correctness of the affected built-in reasoning facilities, but without losing any of the correct information previously introduced into the OB. For example, when moving bed-settee and its instance b-s1to the right position in the hierarchy in figure 4.5b, none of the instance's attributes, values, and constraints, received because it is a furnishing, will get lost.

4.2 The Language KOALA

The language KOALA (KRISYS Object Abstraction LAnguage) provides for the user, application, or OB-designer an abstract, functional view of the OB, defined by means of the operations TELL and ASK to respectively assert the truth of (pieces of) knowledge within the OB, thereby provoking modifications in it, and to retrieve information. Thus, it offers only two operations for OB access: ASK for querying and TELL for specifying state transitions (i.e., OB modifications).

Syntactically, both statements are composed of two parts (figure 4.6). In the projection clause of the ASK-statement, the user specifies the kind of information about the qualified objects that is to be retrieved. In the assertion part of the TELL-statement, the sentences to be asserted are specified. Finally,

the selection clause is used either to qualify information to be retrieved by an ASK-statement or to restrict the application domain of the assertions of a TELL-statement.

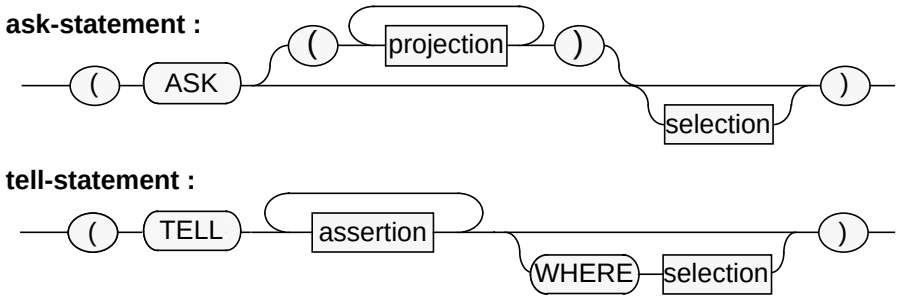


Figure 4.6: The syntax of ASK and TELL.

The assertions and the conditions in the selection clause are both expressed as formulas based on *predefined predicates* and *functions* that embody the semantics of the KOBRA object model. For constructing formulas from predicates, the logical connectives NOT, AND, and OR as well as existential and universal quantification may be used. Operands of predicates or functions can be constants or variables, but also functional terms with appropriate result types, therefore enforcing the orthogonality of the language [Da84]. (In our examples, names of predefined predicates and functions are always written in capital letters.)

Obviously, we have decided not to choose an SQL-like syntax for our language. The view provided by our language is, in some sense, similar to the one presented in [LB86, BFL83] for KRYPTON, which, however, was not developed for an OODBMS context and is therefore not suitable for large applications. Their approach presents at least two major drawbacks: firstly, assertions may only add information to the OB in a monotonic fashion, i.e., asserted facts cannot be removed; and secondly, the user is restricted to yes/no-queries, i.e., he is not able to retrieve or select OB contents (e.g., objects). KOALA, on the other hand, allows changes and deletion of existing knowledge as well as the selection of qualified information.

A similar approach was also taken for the language TellAndAsk of the XPS-tool KEE [FK85]. Let alone the fact that the TellAndAsk language does not provide projection of information and that modifications have to be accomplished with two distinct operations (ASSERT and RETRACT) instead of one TELL statement, the main differences with respect to KOALA (apart from syntactical ones) lie in the expressiveness of the two languages.

Because this section can only give a brief overview of KOALA, a more elaborate comparison with the above approaches is beyond the scope of this paper. A more detailed description of KOALA can be found in [DLM90].

4.2.1 Language Features

Set-Oriented Retrieval

In KOALA, the simplest ASK-statement is the one without a projection clause. Such an operation specifies a condition to be evaluated by the system resulting in a boolean value. In order to allow not only the specification of yes/no questions, set-oriented retrieval of information is provided through the use of variables within the condition (e.g., '?room' in the query below). These variables, which have to be specified in the projection clause as well (printed in **bold** letters in the first example below), are then instantiated with OB contents which satisfy the stated condition. For example, the query below retrieves the orientation and sides of all bedrooms belonging to the living-area.

```
(ASK   ((?room SLOTS orientation sides))
       (AND  (IS-INSTANCE ?room Bedroom *)
             (EQUAL living (SLOTVALUE belongs-to ?room))))
```

This is accomplished by restricting the instantiations of the variable '?room' to the instances of the class 'bedroom' having 'living' as a value of the slot 'belongs-to'. The * as the last argument of the predicate 'IS-INSTANCE' specifies that instances of all classes in the subclass hierarchy of 'bedroom' also have to be considered. In the projection part, the result of the query is restricted to the slots 'orientation' and 'sides'.

Implicit and Explicit Joins

In KOALA, it is very easy to reference objects via relationships between objects across several stages since terms may be nested. We may, for example, select the schema names (without any additional attributes) of all furnishings having 'South' as the orientation of the rooms which they are contained in with the query

```
(ASK   ((?furnishing))
       (AND  (IS-INSTANCE ?furnishing Furnishing *)
             (EQUAL South (SLOTVALUE orientation
                                (SLOTVALUE contained-in ?furnishing)))))
```

The nesting of terms is based on the fact that object identifiers (represented by the schema names) may occur as values of attributes. Compared with the relational model, nested terms can be viewed as implicit joins [Ki89],which are established via object references. Explicit joins (based on comparison operations) may be expressed through queries containing more than one variable. The example below qualifies all instances of rooms (?room) together with the furnishings contained in the rooms (?furnishing) and the size of the furnishing.

```
(ASK   ((?room)(?furnishing SLOTS size))
       (AND  (IS-INSTANCE ?room Room *)
             (IS-IN ?furnishing (SLOTVALUES contains ?room))))
```

Result of a KOALA Query

The result of the above query differs significantly from an equivalent SQL-query, where the result would be a table of joined tuples. In KOALA, however, a set of lists (pairs in the above example) is returned, containing information about the variable instantiations associated through the selection condition, thereby preserving the logical relations between them. This has the advantage, that no object-generating, but only object-preserving operations are used for answering a query (i.e., no new objects are created in order to reflect a query result, not even for 'joins'!). As a consequence, KOALA lacks the closure property usually found in algebraic approaches for object-oriented or relational query languages.

In KOALA, queries may yield not only homogenous, but also heterogenous results. Consider, for example, the first query presented in this section. We might as well specify in the projection clause, that not only the slots 'orientation' and 'sides', but the complete schema specification should be given as a result. Since additional slots may have been added in subclasses of 'room' one may easily have result tuples of different structures in the query result, depending on which subclasses the instantiations of '?room' actually belong to.

More complex queries than the ones presented above may also involve negation, disjunction and quantification of variables. A more detailed presentation is beyond the scope of this paper, but it should nevertheless be emphasized, that the semantics of KOALA is, like in most DB query languages, based on the closed world, domain closure, and unique name assumptions [GMN84], which is especially important for the evaluation of negation and quantification.

Set-oriented Changes with Multiple Assertions

The TELL statement may be used to modify the contents of a OB. Set orientation is easily achieved through correspondence of variables in the selection and assertion parts. For example, the statement

```
(TELL   (IS_IN South (SLOTVALUES orientation ?room))
        WHERE
        (AND  (IS-INSTANCE ?room Room *)
              (EQUAL living (SLOTVALUE belongs-to ?room))))
```

asserts, that all rooms belonging to a living-area should have 'South' as a value of their 'orientation' attributes.

In contrast to relational database languages, which usually restrict the scope of a DB modification to one single table per statement, KOALA allows multiple assertions, which may refer to objects of several classes, within one TELL-statement. In such cases, KRISYS guarantees the truth of all facts asserted after the statement has been successfully performed. If some assertions contradict each other or cannot be asserted for some other reason (e.g., value class or cardinality restriction violation), KRISYS refuses to accept the whole statement.

State-oriented Changes

Note that when specifying assertions, the user does not have to determine the kind of operation to be performed by the system but simply has to describe a state which he wants to achieve. It is the *responsibility of the system to determine the operations (e.g., create, delete, modify) by which this state can be reached*.[1] If, for example, the asserted information is already represented in the OB, KRISYS will carry out no operation at all. For example, the statement

```
(TELL  (NOT  (IS-INSTANCE room1 bathroom 1))
       (IS-INSTANCE room1 bedroom 1))
```

can have the following effects, depending on the current state of the OB:

- If the schema ‘room1’ does not exist, it will be created.
- Schema ‘room1’ will be connected to the class ‘bedroom’, if it is not already a direct instance of that class.
- If ‘room1’ already existed as a direct instance of ‘bathroom’, it will be disconnected from that class.

The last action is taken in accordance with the closed world assumption, which allows negative information to be represented implicitly: The fact that ‘room1’ is not an instance of ‘bathroom’ can not be represented explicitly, but only by achieving an OB state, in which the corresponding positive fact (‘room1’ is an instance of ‘bathroom’) can not be found. Such a state can be reached by removing ‘room1’ from class ‘bathroom’.

The specification of changes in a state-oriented manner *sets one free of having to know the current state of the OB when specifying TELL-statements*. Such a feature is (above all) important for supporting rule processing. KOALA was developed as an interface for KRISYS as well as a language with which the OB-designer specifies rules. However, the kind of operation to be performed by a rule depends on the state of the OB at the time of its activation and is usually not predictable at encoding time of the rule. Consequently, it is necessary to abstract from the state of the OB when specifying rules. For this reason, KOALA follows the state-oriented approach for OB modifications, thereby requiring only the description of a new OB state and not the specification of an operation. So, the definition of new rules may then be performed completely independent of what is going to happen during an inference process.

4.2.2 Accessing Meta-Information

It has already been emphasized in an earlier section, that KRISYS supports not only the maintenance of the OB, but also the design process of OBs and applications. For these reasons, KRISYS eliminates the difference between meta- and ‘usual’ information. Consequently, KOALA was designed to support the OB-designer in his work of building and restructuring the entire OB. Predicates and terms provided by KOALA may be used for retrieving and modifying all kinds of OB contents including meta-information.

---

1. Please note that the language constructs for specifying assertions are only a subset of those available for selections, otherwise it would be impossible to guarantee that a definite OB state can be always reached.

The example shows, how the class hierarchy of a OB can be extended and reorganized within a single TELL statement: A new class ‘livingroom’ is created as a subclass of ‘room’, with a new attribute ‘number-of-windows’. Additionally, all rooms belonging to a living-area are made instances of the newly created class.

```
(TELL   (IS-SUBCLASS livingroom Room 1)
        (IS-IN number-of-windows (INSTANCE-SLOTS livingroom))
        (IS-INSTANCE ?room livingroom 1)
 WHERE
        (AND  (IS-INSTANCE ?room Room *)
              (EQUAL living (SLOTVALUE belongs-to ?room)))))
```

Note that at the time of these modifications, consistency checks and built-in reasoning facilities are dynamically activated. In this case, the system ensures that the class hierarchy does not contain any cycles and that the rooms correctly inherit the new attribute of their class. It is also possible to delete attributes in the same way if the achieved state of the OB does not violate the constraints inherent to the object model. For example, it is not allowed to block inheritance by deleting an inherited attribute of a class.

Additionally, predicates and terms referring to meta-information may also be used in combination with any other predicate in order to construct flexible and powerful retrieval operations. For example, DB query-languages usually require an exact specification of the type of qualified objects (e.g., in the FROM-clause of an SQL statement). In KOALA, objects may be selected without a type specification. The qualification criteria may be specified according to any kind of (meta-) information. Also, if one wants to find out which objects are in an arbitrary way ‘related’ to some object (e.g., my-living-room), a query would be necessary, where even the name of the attribute used for qualification is not specified. In KOALA, such a query may be formulated, because variables can also be used for names of attributes:

```
(ASK    ((?X SLOTS ?S))
        (AND (IS-SCHEMA ?X)
             (IS-IN ?S (OWN-SLOTS ?X)
             (IS-IN my-living-room (SLOTVALUES ?S ?X))))
```

The above query selects all objects, regardless of a type (“IS-SCHEMA ?X”), which contain arbitrary attributes (“IS-IN ?S (OWN-SLOTS ?X”)) having ‘my-living-room’ as one of its values (IS-IN my-living-room (SLOTVALUES ?S ?X))’). Additionally, the variable ?S, which contains the names of the qualified slots, is used in the projection clause in order to achieve a qualified projection, meaning that the result will contain for each object exactly those slots (probably more than one), through which the connection to ‘my-living-room’ is established. This statement could probably be described in SQL-like notation as

```
SELECT      (attributes qualified by predicates)
FROM        *  (or qualified relation)
WHERE       *  = value.
```

4.2.3 Other Important Issues

Some important issues are beyond the scope of this paper, but should nevertheless be briefly mentioned. First of all, we could not present all the language constructs of KOALA. There are additional func-

tions and predicates for expressing transitive closure and generalized transitive closure [DS86]. Also, method calls may be included in KOALA statements, and, vice versa, KOALA queries can be embedded in method code. Additionally, we would like to mention that, although KOALA itself lacks the closure property, the query evaluation process is based on an algebra in order to allow efficient set-oriented processing and optimization.

5. Comparison of the Models and Languages

When comparing COCOON with KRISYS, one easily realizes differences that go back to the overall objectives of both models. The KOBRA model and the query language of KRISYS, KOALA, are more expressive (due to concepts like association, aggregation, rules, more integrity constraints, demons, and recursive queries) and do not separate the database schema from its instances. The reason for the latter is, that KRISYS was not only developed as a database system, but also as a design tool that supports the prototyping process. However, prototyping is beyond the scope of the COCOON model. Instead, COCOON supports only few essential concepts that allow efficient processing of objects (like static type–checking, and a query algebra).

In the following we discuss how concepts of each model are reflected in the other one, and we show the differences between the models and query languages.

5.1 The Object Models

5.1.1 Object Types and Collections

Both the COCOON and the KOBRA model support the definition of object types and/or collections, which can be organized as hierarchies (or networks), and are closely related to the notions of object structure (and behavior), inheritance, set membership and inclusion, as well as membership conditions[MMM92]. Also, both systems allow objects to belong to several types and collections at the same time and support multiple inheritance.

The COCOON model explicitly distinguishes the concepts of types, which include state–independent information (like signatures of functions), and classes (i.e., state dependent collections), because of the following reasons: First, the separation allows static type–checking. Secondly, due to this separation the definition of different collections with the same type becomes possible (e.g., a class of all rooms, and a class of the rooms in my flat). Thirdly, the case that the subtype–relationship holds in the opposite direction to the subset–relationship (which occurs already between classes and query results, if the queries contains a projection and a selection, e.g., see Figure 3.2) can be taken into account. Without this separation the class and the query result cannot be related to each other.

In KRISYS the notion of an object type is not supported by a separate concept, but is integrated with the notion of its extension into one uniform concept, the class. Type checking is performed dynamically at run time, which is of course less efficient, but allows more flexibility than static type checking. Although KRISYS automatically associates with an object type a default collection, i.e., its extension, there is still

a distinction between classes and arbitrary collections of objects, which can be represented as sets using the association concept. Sets can be associated with classes by restricting the possible elements of a set to instances of the class. One may, for example define an object 'rooms-in-my-flat' as a set of 'rooms' (see Figure 5.1).

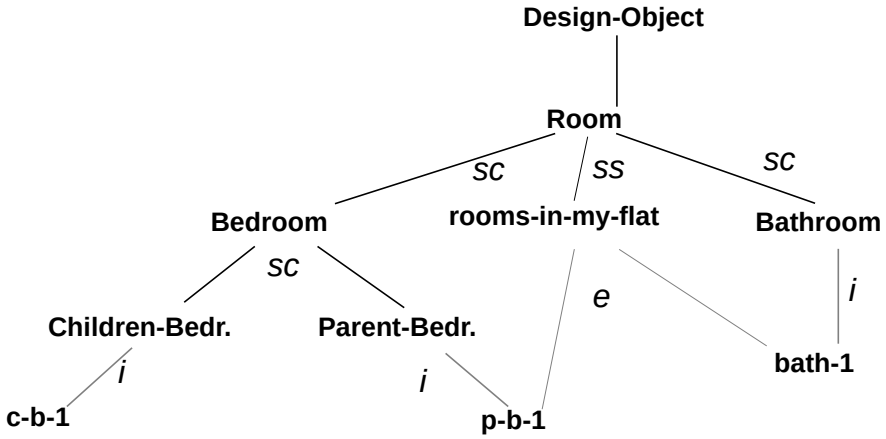


Figure 5.1: Defining arbitrary object collections with the association concept

5.1.2 View Definitions

COCOON supports views, for the same reasons as in the relational model. Due to the closure property of COOL any query (except *pick* and *extract*, which do not return sets of objects) can be used as a view definition. The only difference between views and classes is that the extent of views is derived by the system. Therefore, both can be used in subsequent query or update operations. Particularly, views become part of the global schema and are automatically classified into the class–hierarchy.

For example, a view that contains the functions *name, neighbors, size* of large rooms can be defined by the following expression:

define view *LargeRoomV* **as project** [*name, neighbors, size*](**select** [*size > 16*](*RoomC*))

The position of the view in the class hierarchy is determined by a classifier, which derives the shaded arrow in Figure 5.2..

The concept of view definitions is not directly supported in KRISYS, but the association concept may be employed in a similar way. Membership stipulations may be used as an intensional description of a set of objects, which is roughly equivalent to the select–views of COCOON. In order to define large rooms in KOBRA, one would define the set 'large–rooms' using the association concept (see Figure 5.3), and define a membership stipulation (size > 16) for it, which corresponds to the select part of the view definition in Figure 5.2 The project part, however, is not supported in this definition. Additionally, positioning the set within the generalization or association hierarchy is not carried out by the system, but has to be performed explicitly by the user.,

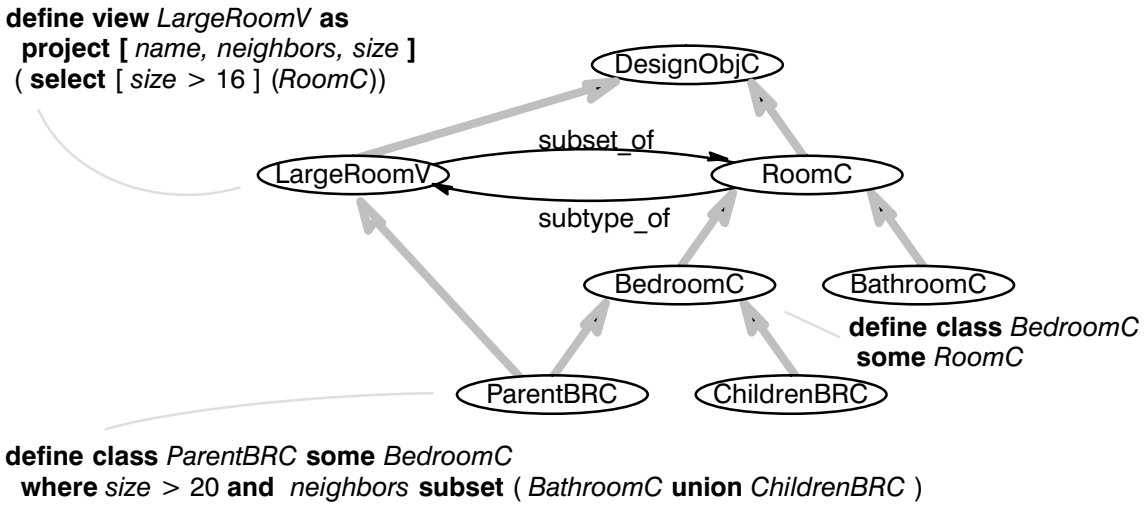


Figure 5.2: Result of the classification of the view *LargeRoomV*

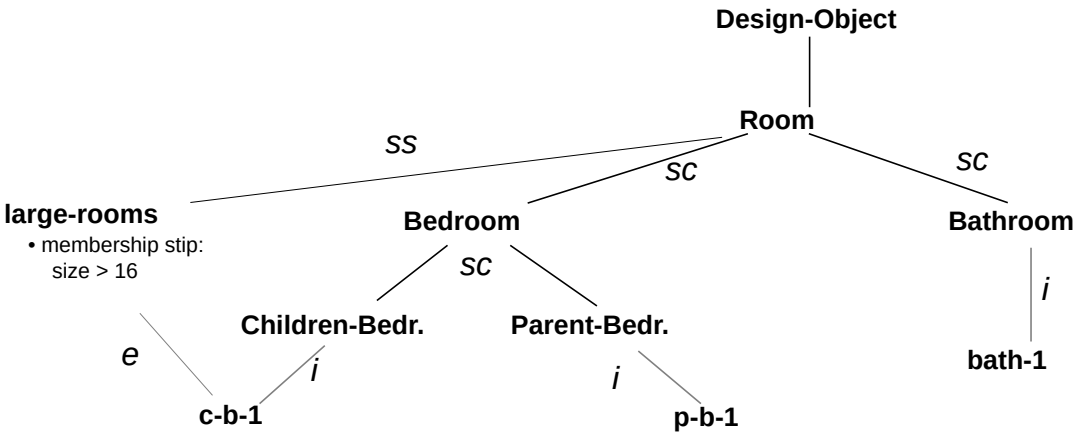


Figure 5.3: Using membership stipulations for defining ‘selection-views’

5.1.3 Object Base Design Support

In COCOON, we find a strict separation of the database design and the operation phase, and only the operation phase of an application is supported by the data management facilities of the system. Therefore COCOON regards meta–information (i.e., the OB schema) as rather stable, and concentrates on object management tasks. Nevertheless, there are considerations to introduce more flexibility, particularly on the application of the standard COOL operations for realizing schema evolution [TS92]. However, the storage structures of COCOON are optimized for bulk operations on objects for a relative stable storage schema.

KRISYS, on the other hand, can be seen not only as an object base management system, but also as a modeling tool for object bases. Meta information is integrated into the OB and can be flexibly changed or extended by the OB designer in an interactive fashion without loss of information (w.r.t. the OB 'instance'). As already sketched in Section 4, the built–in reasoning facilities of the abstraction concepts are the basis for this dynamic modeling support. Changes of (as well as queries on) meta information

are supported within the framework of KOALA, therefore providing powerful means for incremental OB development and set-oriented reorganization. For example, the statement

```
(TELL   (IS–SUBCLASS livingroom room 1)
        (IS–INSTANCE ?room livingroom 1)
 WHERE
         (AND  (IS–INSTANCE ?room Bedroom *)
               (EQUAL living (SLOTVALUE belongs–to?room))))
```

performs a typical modeling activity, defining an additional subclass of 'rooms' (i.e., 'livingroom'), and reorganizing the existing instances of 'rooms' w.r.t. the new class.

The support of modeling activities has, of course, consequences on the way the KOBRA model is mapped to the PRIMA system, which will be explained in Section 7.

5.2 The Query Languages

COOL and KOALA differ in their expressiveness (besides obvious syntactical discrepancies), which has already been outlined above and will not be further elaborated here. Instead, we point out in the following several issues related to query language semantics in order to compare the two languages w.r.t. their fundamental properties.

5.2.1 Closure Property

When considering the result of a query, it is important whether the query 'leaves' the object model in the sense that it does not return objects of the model, but only information about objects to the application, or whether the query is closed w.r.t. the object model. The latter is indispensable for views and subqueries.

As an algebra, in which nesting and composition of query operations is inevitable, all query operations of COOL - except the operation ***extract***, which is used for the presentation of values to users (see below) - are closed against the model COCOON. This is, operations are applied to and result in typed sets of objects.

In KOALA, an ASK statement generally leaves the model, i.e., the closure property does not hold. However, the instantiations of query variables that might appear in the selection part of queries, can be used as an input for update operations (TELL statement) and during the execution of rules, which are also specified using KOALA. In order to use query results for further operations, they can be inserted into the OB (e.g. as a temporary result) by an appropriate TELL statement. The fact that the query variables preserve the model semantics is also sufficient for guaranteeing an operational closure for the query algebra of KOALA, which is fundamental for query evaluation and optimization.

5.2.2 Object Preserving Operations

During query evaluation, a sequence of operations is performed, which produce new results out of existing temporary or base information. Contrary to relational systems, that are strictly value–based, it is important to distinguish OODBMS operations which generate new objects from their input and opera-

tions that preserve the objects in their identity. Since object identifiers are commonly used as object references for example in attribute values, the use of object generating operations may cause severe problems in update operations, and if the retrieved objects are temporarily stored and used for subsequent queries.

Object preserving semantics is the central concept in COOL that allows a straightforward view definition facility. Because the objects contained in views are the *same* ones as in the base classes, updates applied to views can easily be propagated to the base classes and vice–versa [SLT91]. Additionally, object preservation allows to position views close to the base class in the class hierarchy (e.g., see Figure 5.2).

In KOALA, an ASK statement leaves the model, therefore satisfying the object preserving semantics in a trivial form. But also the instantiations of query variables are treated in an object preserving manner. No objects are created during the evaluation of an ASK statement or the selection part of a TELL statement. Even join–like operations do not create new objects, but only cause a recording of the association of objects to be joined.

5.2.3 Presentation of Query Results

In order to present query results to users or programming languages that can not use the OID, COOL offers the value generating operation *extract* that generates (nested) relations [SS90b, SLR+92]. This is, similarly to the relational projection, a list of functions, which might include nested ***extract*** statements, defines the schema of the result relation.

In contrast to the ***extract*** operation, the result of an ASK statement of KOALA might be a heterogeneous value. For example, a query might request the complete object descriptions of all instances of a certain class (e.g., 'furnishings'), where not only direct, but also transitive instances (i.e., instances of subclasses) shall be included. If no special option in the projection clause is given, the query yields the complete objects (including additional attributes according to subclasses). Otherwise, the result is made homogeneous. Note, however, that heterogeneous results may be desired by the user or application.

5.2.4 Update Operations

COOL provides generic update operations (e.g., insert, delete) for changing the contents of an OB. The advantage of the generic update operations is that they have a deterministic semantics, are defined with respect to all model–inherent integrity constraints, and that they can be used for the implementation of more complex update methods [LS92]. Thus, more sophisticated application–specific constraints can be implemented in terms of methods that contain generic update operations as elementary operations.

In contrast to COOL, where changes are required by giving the exact operation to be performed, KOALA supports the notion of state–orientation, meaning that the user does not have to determine the kind of operation to be performed, but simply describes a state to be achieved. It is then the responsibility of the system to determine the exact operation necessary to reach this state. Changes are therefore not specified in terms of update, insert, and delete operations, but are accomplished by a general TELL statement.

6. Mapping COCOON to DASDBS

In this chapter we describe the mapping of the COCOON object model to the DASDBS kernel system, using the architectural example.

DASDBS was chosen as the storage system, because of its support for complex storage structures. First of all, the support of nested relations allows for the storage of hierarchically clustered data. That is, we have hierarchical access structures with an arbitrary level of nesting, as well as the opportunity to define nested join indices. This can be very useful to store COCOON objects and the ”relationships” between them.

Second, the DASDBS interface offers powerful data retrieval and manipulation operations. The system has a set-oriented, algebraic interface, with efficient operations on complex objects. All operations performed in the kernel are executed in a single scan through the data [PSS+87, SPSW90]. Operations that are not single scan processible are performed outside the kernel, in the higher-level query processor. This query processors ability of processing on complex nested structures, further improves processing, like for reassembling complex object structures [RRS91]

The overall architecture of the COCOON implementation on top of the DASDBS kernel system is shown in Figure 6.1. The two aspects of this architecture, structure mapping and operations mapping, are realized in the physical design tool and the query optimizer, respectively. Physical design uses the COCOON schema, statistics about cardinalities and distribution, and a description of a transaction load to propose a good internal storage structure expressed in terms of nested DASDBS relations. At transaction processing time, the optimizer has to translate COOL operations down to operations on these physical NF^2 structures. The execution plans generated consist of physical NF^2 algebra operators, some of which, such as joins, are implemented in the high-level query processor, others are DASDBS kernel calls. In the sequel, we elaborate on each of these two aspects separately..

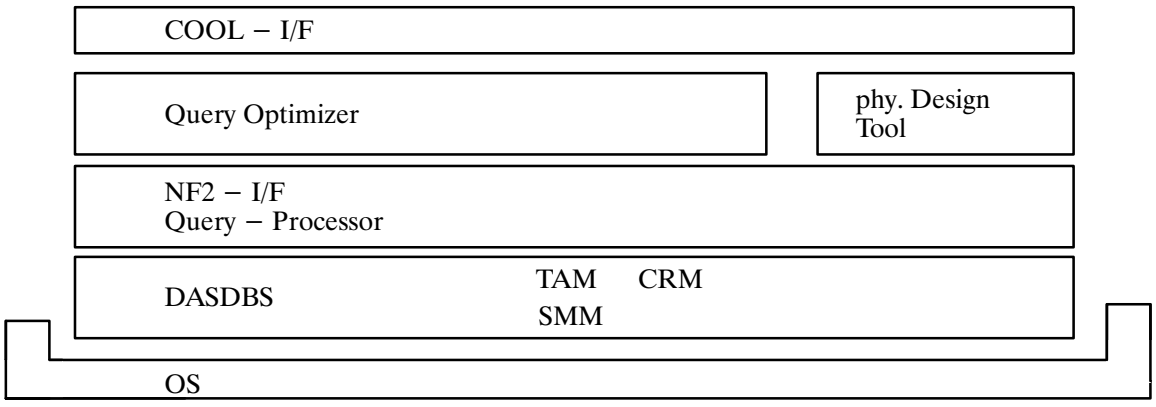


Figure 6.1: The COCOON − DASDBS Architecture

6.1 Physical Database Design

In this section, we discuss the mapping of COCOON schemas to nested relations (DASDBS) [Sch93]. That is, given a COCOON database schema, what are the alternatives for the internal DB layouts, and further, given a transaction load, which internal DB layout results in the least overall cost of transaction execution. This optimization task is performed by the physical design tool.

Because COCOON (like almost all OODBMS) offers a variety of structuring capabilities to model complex objects, it is not at all trivial to find good, that is, efficient, storage structures that support the variety of operations on objects reasonably well. Objects may be hierarchically composed of subobjects, several objects may share common subobjects, objects may appear as (attribute) values of other objects, different objects can be related to each other by functions, methods, or relationships. Type (or class) hierarchies introduce another dimension of object interrelation: an object of one class also ‘‘appears’’ in all its superclasses; again, with multiple inheritance, this need not be a strict hierarchical inclusion. Computed values (attributes, methods) may be used to derive, rather than store, data that are associated with objects.

6.1.1 Alternatives for Physical DB Design

In order to explain the alternatives for mapping COCOON database schemas to nested relations at the physical level, we proceed by stepping through the basic concepts of the COCOON object model, and showing the implementation choices. Since the choices for each of the concepts combine orthogonally, this spans a large decision space that is later on investigated by the physical database design tool (next section).

Implementing Objects

According to the object-function paradigm of COCOON, an object itself is sufficiently implemented by a unique identifier (OID), that is generated by the system. All data related to an object in one way or the other will refer to this identifier (see below). We denote for each object type *A*, attributes of internal relations containing the OID of objects of type *A* by *AID*.

Implementing Functions

In COCOON, functions are the basic way of associating information (data values or other objects) to objects. In principle, we can think of each function being implemented as a binary relation, with one attribute for the argument OID and the other for the result value (data item or OID). In case of set-valued functions the second attribute will actually be a subrelation of unary subtuples, containing one result (OID or data value) each. So, in principle, each single-valued function as well as each multi-valued function would be implemented in a binary relation. For example, in our architectural design application, the functions *size* and *furnishings* of *Rooms* would result in two relations, *Room_size(RID, size)* and *Room_furnishings(RID, furnishings(FID))* respectively. Obviously, there are some choices (such as, do we really store each function in a separate binary relation or do we combine several of them into a ‘‘wider’’ relation?), and also some more subtle alternatives (such as, shall we include physical pointers?). The decision

space as far as function implementations are concerned includes the following alternatives in our current approach:

Bundled vs. Decoupled: Each function defined on a given domain object type might either be stored in a separate (binary) relation as shown above: the *decoupled* mode. Alternatively, we can *bundle* functions together with the relation implementing the type.

In the example above, the bundled implementation of *Room size* and *Room furnishings*, would yield the following type table : *Room(RID, size, furnishings(FID)).*

Logical vs. Physical Reference: A function returning a (set of) object(s), not (a) data value(s), can be implemented by storing just OIDs (*logical reference*) of result objects or by including a TID (*physical reference*) as well. Continuing on the above example (bundled), inclusion of physical references for the *neighbors* function, would result in: *Room(RID, size, furnishings(FID, @F)).* Notice the naming convention: (physical) reference attributes have @ as a prefix.

Oneway vs. Bothway References: A function can be implemented by a forward reference only (*oneway*), or it can be implemented with backpointers (*bothway*). Notice that COCOON includes the specification of *inverse functions*. If an inverse functions is defined in the conceptual schema, then the ‘‘backward’’ reference is present anyway. Therefore, this option is only considered for functions that have no inverse in the object schema. Decoupled functions with backpointers result in ‘‘join indices’’. For example, in case the *contained_in* function of *furnishings* would not be present in the conceptual schema, we could nevertheless decide to implement it, to have the *contains* function of *Rooms* supported with backpointers as well.

Reference vs. Materialized: Functions returning (possibly sets of) objects, not data values, can be implemented by the various forms of references discussed up to now. Alternatively, however, we can directly *materialize* the object-tuple(s) representing the result object(s) within the object-tuple representing the argument object. This is a way of achieving physical neighborhood (clustering).

In our example, the decision to materialize the *furnishings* function of *Room* objects would generate a nested type table that contains the type table for *furnishings* as a subrelation: *Room(RID, size,... furnishings(FID,name, ...)).* Obviously, we need no backward references in this case. Furthermore, this alternative is free of redundancy only if the materialized function is *1:n*, that is, it’s inverse is single-valued.

Computed vs. Materialized: Finally, an additional option is to materialize derived (computed) functions. Assuming that some function can be computed, we could nonetheless decide to internally materialize it, if retrieval dominates updates to the underlying base information significantly. The more retrieval dominates updates, and the more costly the computation is, the more likely is the case that materialization pays off. For example, the *size* function of *Rooms* is derived from the actual geometry of the Room. But computing the *size* incurs quite some effort and if object shapes rarely change, materializing the *size* function clearly is a good strategy (see also [KM90a,KM90b]).

Implementing Types, Classes, and Inheritance

The COCOON model separates between types and classes, this results in having two inheritance hierarchies: one between types (organizing structural, function inheritance), and one between classes (organizing set inclusion). This raises the question whether we do the design for types of objects, or for individual classes, and how to implement the inheritance hierarchy. Since classes are always bound to a particular (member-) type, physical design for types is the larger grain approach, whereas design for individual classes would be the finer grain approach (remember, there may be more than one class per type).

Currently, we do the physical design on a type basis, that is, all objects of a given type are physically represented in the same way (even if they belong to several classes). Classes are implemented as views over their underlying type table. This results in the following choices w.r.t. types, classes, and inheritance:

Types: Each object type is mapped to a type table with at least one attribute, containing the OID. Additional attributes are present in case of any bundled functions and/or materializations of object functions. The type table *T* may itself be a subrelation of some other table, if type *T* was materialized w.r.t. a function returning *T*-objects.

Classes: Each class *C* is implemented as a view over its underlying type table. If the class is defined by a predicate (''all''-classes and views in COCOON), this predicate is used as the selection condition. If the class is defined to include manually added member objects (''some''-classes in COCOON), the underlying type table is extended by a Boolean attribute *C* that is set to true, if the object is a member of this class *C*.

Inheritance: Subtyping is implemented by having one type table per subtype. Three possibilities are considered:

- an object-tuple is included in each supertype's table. In case there are any bundled or materialized functions, these are not repeated in the subtypes' tables. In this case, object-tuples in subtype tables might optionally include physical references to supertype tuples.
- an object-tuple is included only in one type's table, that of the most specific subtype. In case of any bundled or materialized functions in supertypes, these are also included in the subtype's table.
- an object-tuple is included in each supertype's table. In case of any bundled or materialized functions in supertypes, these are also included in the subtype's table.

6.1.2 The Default Physical Design

In order to have a starting point for both, the physical database design tool described in the next section, and the implementation of COCOON on top of DASDBS, we have identified a default physical design that includes the following choice of implementation strategies:

Functions: All functions are bundled with their type table. Object–valued functions are implemented as

references, with physical references and backpointers. Multi–valued functions become subrelations.

Inheritance: Objects are present in all supertype tables, inherited functions are not repeated in subtype tables. No backpointers to supertype tuples are included.

The default physical design for our architectural example world, given in the beginning (Figure 2.1), is shown in Figure 6.2 . Further, we show an alternative physical design, where the object valued function *furnishing* is materialized, in Figure 6.3. Both designs will be used later on, when describing the query optimization task.

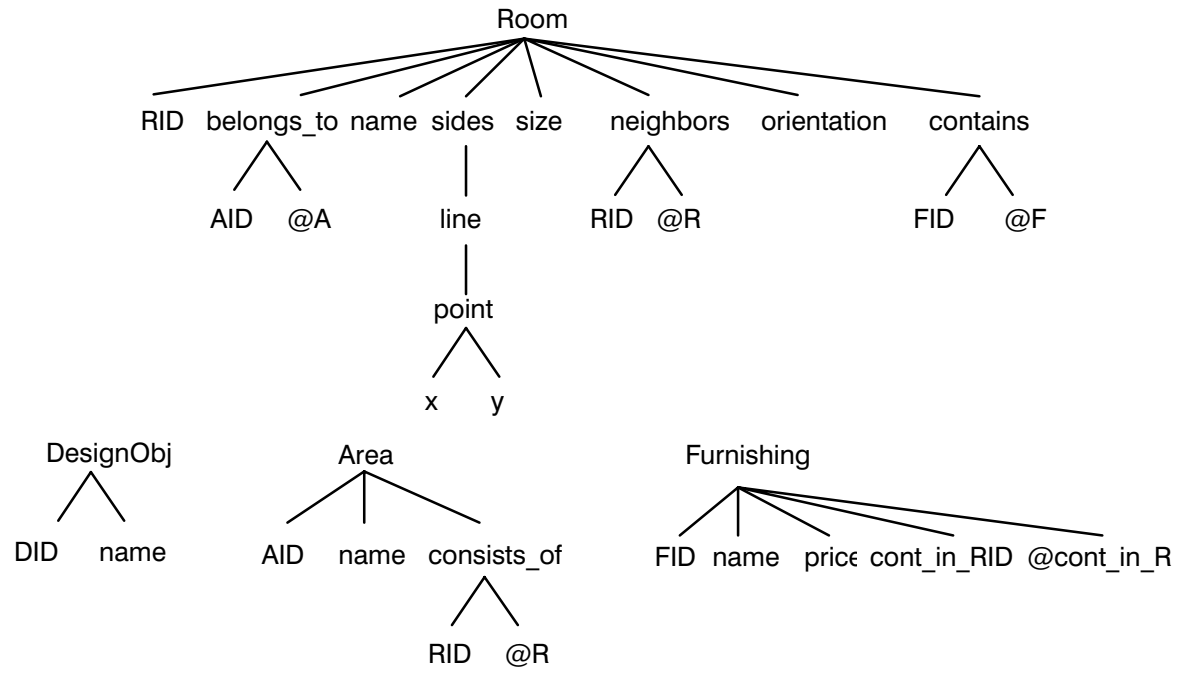


Figure 6.2: Physical Design I of our Example DB−World

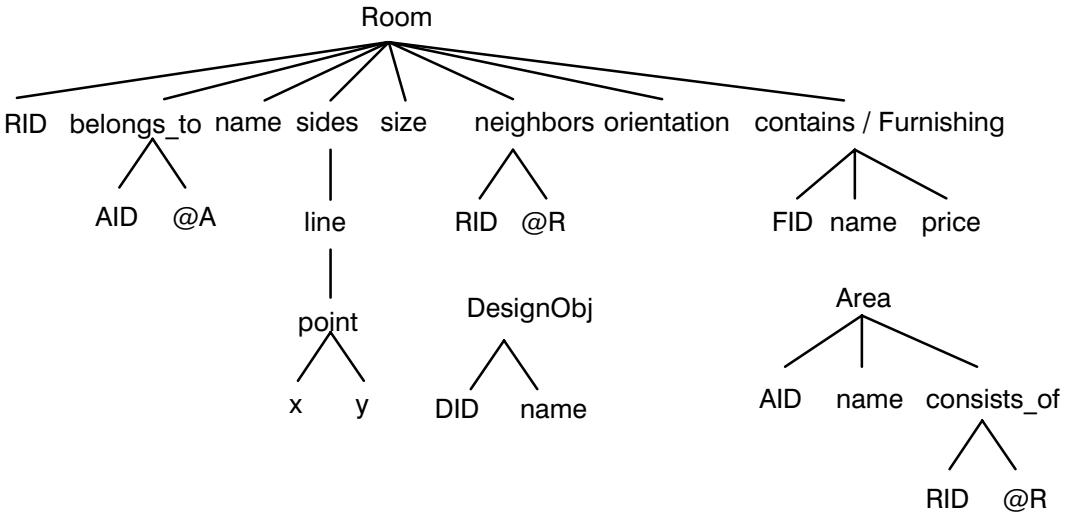


Figure 6.3: Physical Design II of our Example DB−World

6.1.3 The Physical Design Tool

In this section, we describe our preliminary physical database design tool (DBDesigner) that considers some of the above alternatives and produces an internal nested relational schema for use with DASDBS, derived from a conceptual COCOON schema, a load description, and a cost model. The system was developed in a sequence of master theses [Gro91, Goe92, Sto92]; it is implemented in PROLOG.

The transaction load (which may be estimated, observed, or ‘‘guesstimated’’) is given to DBDesigner as a collection of abstractions of COOL operations together with relative or absolute frequencies each. For example, selections are specified by the attributes in the predicate, the general form of the predicate and the estimated selectivity. Extract operations are described basically, by specifying what parts of the objects are read for output.

Example: If 25% of the load consists of extracting *Room* data together with (nested) *Furnishing* data, the load description will contain the following line:
/.25/extract[name, size, extract[name, price](contains)] (Room);

In order to compute the cost of operations, the design optimizer should actually cooperate with the query optimizer of the execution engine. In the current development phase of the COCOON-DASDBS mapping, however, this part is not yet completed. Therefore, DBDesigner uses its own set of statistics and cost formulae. The statistics used by the physical designer include cardinalities of types (how many objects of that type are in the DB?), (average) cardinalities of set-valued functions, and (average) sizes of all atomic values [Gro91]. The cost model is a simplified version of a cost model for DASDBS operations developed earlier [BD87, Pau88].

The optimization is performed in the following way: First, from the given load description a ‘‘transaction graph (TG)’’ is constructed, which is later used for enumerating the design alternatives. The TG consists of vertices representing the types in the conceptual database schema, directed edges connect types if there is a ‘traversal’ in the corresponding direction in the transaction load. Continuing the example given above, the description of the extract statement would result in an edge from node *Room* to node *Furnishing* in the transaction graph. Other possibilities for such traversals are selections and ‘information flow’ in update statements. Further, each edge in the TG is assigned a weight. This weight represents the accumulated traffic across this link, that is, from the load description we accumulate the frequencies of all operations that incur the traversal represented by the edge.
After the construction of the TG, the second step, that is, the optimization is performed. Starting from the default physical design (see 6.1.2), the optimizer selects in a branch-and-bound fashion the most promising (that is, heaviest) edge from the TG and tries to improve performance by, e.g. materializing the corresponding function (physical clustering) or by repeating the inherited attributes in the subtype’s object-table.

Experiences with the DBDesigner, (with rather small example databases, with only a modest complexity in the transaction load) have been carried out. With the small test cases, as expected, the performance was good, PROLOG has not (yet?) turned out to be a big penalty. Extensions to DBDesigner, to allow dynamic modifications in either the transaction load or the database schema have been added [Goe92].

The consideration of redundant storage schemas, particularly for the materialization of views is investigated in [Sto92]. Up to now, no access path are suggested by DBDesigner. A possibility to restrict the level of nesting will be added to allow the DBDesigner for storage managers that can not support arbitrarily deep structures (such as PRIMA or Oracle with ”Clusters”, both of which allow for 2-level nested structures only).

6.2 Query Optimization

In this section, we discuss the transformation and optimization of queries that are given to the system in terms of the COCOON database schema [RS93]. It is the task of the query optimizer to map these COOL queries down to the physical level by: (i) transforming them to the nested relational model and algebra as available at the DASDBS kernel interface, and (ii) select a good (if not the best) execution strategy.

Because COCOON’s query language, COOL, is pretty similar to a nested relational algebra, a straightforward transformation from COOL expressions down to a nested relational algebra expression against any fixed implementation on the internal level (e.g., the default physical design) is rather easily done. Complications arise from the fact that the mapping of data structures is very flexible, and that, depending on the chosen design, operations have to be optimized substantially.

We have investigated two competitive approaches to query transformation and optimization. The first one is a purely algebraic one, comparable to what we did with the relational to nested relational mapping [SPS87,SS90a]: COCOON classes would be defined as ‘views’ over the stored nested relations, COOL queries would be transformed to the nested relational level by ‘view substitution’, and finally, algebraic transformations within the nested relational algebra could be applied, so as to eliminate redundant subexpressions. Quite a few redundant joins would have to be removed in case of materialized functions (hierarchical clustering). This has exactly been the problem addressed in [SS90a].

Example 6.1: Given the COOL query

```
Q1 := extract    [  name, orientation, size,
                    extract[ name ] ( neighbors ),
                    extract[ name, price, width, length ] ( contains )
                 ]  ( Room );
```

and assuming the default physical design (Figure 4.2) in the transformation of this query to the NF^2 level, this results in the query

```
Qc := PROJECT    [ name, orientation, size,
                   PROJECT [ name ] ( neighbors ),
                   PROJECT [ name, price, width, length ] ( contains ) ]
   ( JOIN [ Room.contains.FID=Furnishing.FID ]
     ( JOIN [ Room.nbrs.RID=Room.RID ] ( Room, Room ), Furnishing ) )
```

However, if the internal physical design is the one given in Figure , that is the function *contains* is materialized, one join operation can be removed by optimization, resulting in the query Q_i. Notice, that further optimization could be performed, which is not shown here.

```
Qi := PROJECT    [ name, orientation, size,
                   PROJECT [ name ] ( neighbors ),
```

PROJECT [name, price, width, length] (contains)]
(JOIN [Room.nbrs.RID=Room.RID] (Room, Room))

The second approach directly uses the information about the physical database schema in the transformation of a given COOL query into a nested relational algebra representation. No redundant joins are created, that would have to be removed in the following optimization phase. Thus we have a more direct transformation. In Example 6.1 above, this would result in transforming query Q_1 directly onto the physical schema, that is, into query Q_i.

We have chosen this second approach in our current implementation. The optimization phase following the transformation chooses the specific execution strategy, e.g. the ordering of operators, as well as the best implementation strategies. For example, whether a nested loop or a sort merge join is selected. This leads to the following architecture of our query optimization process.

In the first step, the transformation task, we use a ”Class Connection Graph”. We will elaborate more on that in the following. The second step, that is, the optimization phase, we use a rule–based algebraic query optimizer, generated with the EXODUS Optimizer Generator [GD87].

6.2.1 Transforming COOL–Queries onto the Physical Schema

The input to the optimizer is a COOL-query expressed on the conceptual schema, while the transformations apply to execution plans, i.e., on the physical schema. Therefore, we have to do a translation to an algebraic query representation on the physical schema first. To do this, we use a Class Connection Graph similar to the one proposed in [LV91].

After the input query is parsed, the optimizer scans at the same time the query graph (a graph representing the given query) and a physical schema graph. From these, the class connection graph is constructed. A possible physical schema graph is given in Figure 6.5. This schema graph corresponds to our physical design, shown earlier in Figure 6.3. The nodes are files, which may implement
- a single class extension (e.g. A implements Areas and D implements DesignObjects)
- several class extensions (e.g. R implementing Room and Furnishings)
- a part of a class extension, when a class extension is vertically fragmented.

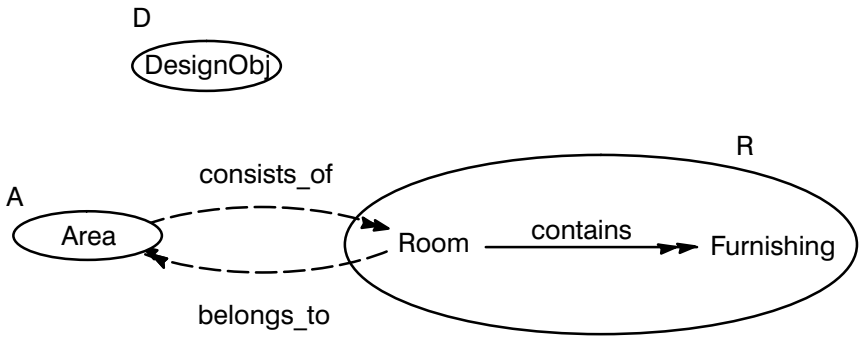


Figure 6.5 : A sample Physical Schema Graph

The arcs denote the kind of function implementation. Solid arcs denote materialized functions (e.g. contains). Dashed arcs are functions stored as references to subobjects inside the instance of the owner object.

Now let us look at the transformation of a sample query. Suppose we have a physical schema graph as shown in Figure 6.5 and the following query:

Query Q2 :

```
extract [ name, orientation, size,
        extract [ name ] ( neighbors ),
        extract [ name, price ] ( select [ name = ”computer” ] ( contains ) ) ]

( RoomC )
```

Figure 6.6 shows the Class Connection Graph constructed for that query. The nodes of the class connection graph represent classes which are affected by the given query. The edges are the functions involved in the query, connecting classes via implicit or explicit joins. There are three different kinds of edges, depending on the physical representation of these functions, i.e. did we store logical OIDs, physical references, or is this object function materialized. In our example, the *contains* function of *Room* objects is materialized, and the *neighbors* function is supported by pointers. The use of the *neighbors* function results in a second occurrence of class Room, here denoted as *Room'*. In case a class extension would be stored vertically fragmented, this would result in having a class for each fragment, all of them connected with corresponding arcs. Further, we see the selection predicates (name = ”computer”) and the printable attributes, which shall be displayed (name, size, and orientation of Rooms).)

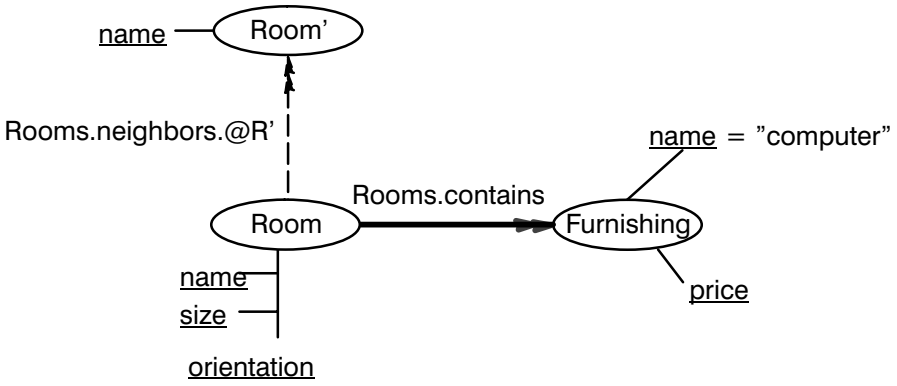


Figure 6.6: A sample Class Connection Graph

A class connection graph represents a given query, without any hints on execution orders, that is, all possible execution plans can be generated from the class connection graph. For example, we have the choice to perform forward or backward traversals, or to start in the middle of a path, as well as to interleave other operations with the traversal (cf. [PB89]).

6.2.2 Optimization of Query Execution Plans

In this section we describe the model of execution offered by the base system (DASDBS kernel and query processor) and how query execution plans are represented. Further, we will give some example queries and demonstrate how they are optimized by our system.

To represent the execution plans, we use processing trees. Two possible processing trees are given in Figure . The leaf nodes, which represent complex operations on one DASDBS relation, are performed (in a set–oriented way) by the DASDBS kernel. Internal nodes are performed it the COOL–specific query processor (in a streaming mode).

Figure 6.7: Two equivalent Processing Trees

Which Potential for Optimization do we have?

Given a query, obviously there are many equivalent processing trees, that is, alternatives to execute the query. These alternatives result from a number of open choices, some of which are listed in the following.

Join Ordering. A sequence of join operations is freely reorderable, this results in many alternatives. In order to reduce the number of orderings to consider, one may exclude the ones resulting in Cartesian products, or restrict the join orderings to the ones resulting in linear join trees, instead of considering all possible, bushy ones (this is the well-known problem studied, for example, in [Sel79, OL88, SG88, Swa89].

The reason for join operations to occur is the following: First, reassembling objects from several relations requires a join operation for each partition. Object partitioning is introduced for example, in the mapping of inheritance hierarchies. Second, each application of an object valued function (which is not materialized) results in a corresponding join. Reordering these joins corresponds to changes to the order of function application, that is whether we do forward or backward traversals, or starting in the middle of a path query [Ber90]. Notice, that these implicit joins may be supported by link fields (see Section 6.1.1), and therefore enable efficient pointer based join algorithms [SC90].

Pushing Selections. Selections may be performed at various times. For example, one could perform a given selection before or after applying an object valued function (move selection into join, or vice versa). Additionally, selections with conjunctively combined selection predicates may be split into

two selections (or vice versa), and the order of selections, as well as the order of the terms in selection predicates, may be changed.

Pushing Projections. Projections, in the same way as selections, may be performed at various times. In addition, the order of applying projections and selections may be changed.

Method Selection. For each logical operation there may be several methods, that is physical implementations. For example, a join operation may be performed by nested loops join, hash join, or sort merge join. In addition, for implicit joins which are supported by link fields, pointer based join versions may be selected. Further, selections may be performed by using indexes, or by performing a relation scan, followed by predicate testing.

Index Selection. If a selection is supported by multiple indices, one has the opportunity to use all, some, or just one of the available indices.

Almost all of the choices given above, can be combined orthogonally, such that the number of equivalent processing trees grows extremely fast with increasing query complexity.

How to find the Optimal Execution Plan

One way to find the optimal execution plan for a given query is to simply generate all possible execution plans, estimate the processing cost, and choose the least expensive one. Obviously the optimizer would be unacceptably slow, if every possibility would be considered. Sophisticated optimizers have smart search strategies, which avoid considering as many as possible alternatives, which will (at least with high probability) not lead to the optimal execution plan. Due to the fact that a query optimizer is one of the most intricate subsystems of a database system, it is desirable not to start an implementation from scratch. Therefore, we decided to use the EXODUS optimizer generator for the implementation of the COOL optimizer. Without going into detail, a cost model to estimate the quality of execution plans, as well as rules to describe possible transformations have to be defined by the implementor of the optimizer, the search strategy is provided by EXODUS.

Obviously the cost model plays an important role in the optimization task. In order to find the cheapest query execution plan, it is necessary to determine the cost of an execution plan. We use cost functions, which have to be defined for each physical implementation of a logical operator, to calculate the processing cost (CPU, I/O and network transfer) as a function of its arguments. The cost for the entire execution plan is calculated as the sum of its subplans, that is, the sum of its methods costs. The cost functions for methods are fairly standard, similar to the ones of System R [Sel79]. They had to be adapted to the nested relational model. These functions are based on information about the stored data, like tuple and page cardinalities for each relation, and the number of distinct values for each attribute. Further improvements, like including information about data distribution, as well as taking indexes into account, are planned for the future.

Rules describing possible transformations of processing trees, that is, of nested relational algebra expressions, are derived from transformation rules known from the (flat) relational model. Roughly, 1NF

rules are applicable in the NF^2 model on relations and on the subrelations, having additional constraints and more complex argument transfer functions. Further, additional (complex) rules have been defined for the nested model [Sch86, Sch88]. The EXODUS optimizer generator was designed with the intention to be extendible, that is, it should support query optimization without making restrictive assumptions about the data model, the query language, or the run time system. So using the EXODUS optimizer is in principle quite easy. Looking deeper into it, it turned out that due to the enhanced complexity of the rule set, in addition with more sophisticated constraints and ”argument transfer functions”, a considerable amount of additional work has to be done by the optimizer implementor.

Example rule: The rule how two project operations can be combined into one project operation, can be formulated in the relational model in the following way:

$$\pi\,[A]\ (\pi\,[A]\ (E)\,) \equiv \pi\,[A]\ (E)$$

(Obviously, A $\subseteq$ B has to be fulfilled, to have a correct expression on the left side.)

This rule leads to several rules model, each having an increased complexity. One resulting rule is given below:

$$\pi\Big[\ldots\sigma_{F_2}(A)\ldots\Big]\pi\Big[\ldots\sigma_{F_1}(A)\ldots\Big]\,(E) \equiv \pi\Big[\ldots\sigma_{F_2}\sigma_{F_1}\ldots\Big]\,(E)$$

Using the notation enforced by EXODUS, this rule is not expressible, rather it has to be formulated in the following way (actually, in the same way as the relational rule):

$$\pi 1\,(\pi 2\,(E)\,) \equiv \pi 1\,(E)$$

But this rule is rather incomplete, the missing parts have to be supplied in so–called support functions, that is, the condition code (to check whether the rule is applicable) and argument transfer functions.

A first prototype optimizer has been implemented [Lae91, Hof92], and integrated [May92] into the COCOON-DASDBS system. COOL queries are passed from the COOL interface to the optimizer, and after translation onto the physical schema and optimization, execution plans can be passed to the query processor to execute the query on the DASDBS kernel. The functionality includes removal of redundant operations, the combination of operations, and select-project-join ordering, for nested relations. Nested Loop-, Nested Block-, and Sort Merge Joins are considered as implementation strategies. After the completion of the implementations of pointer based join algorithms [Wil91], these methods will also be added to the optimizer’s repertoire. Since our goal was to evaluate the advantages and cost of using a complex record (DASDBS) instead of flat record (RDBMS) storage manager, the main emphasis has been on the effects of hierarchical clustering and embedded references; indexes played only a supporting role; future improvements should include indexes as well.

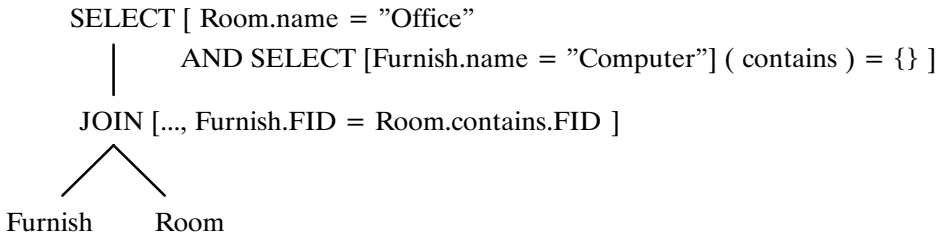
6.2.3 Example Transformation and Optimization

In the following we will give one more example COOL query, and demonstrate how the query is mapped to an execution plan for the query processor, and where performance gains are achieved.

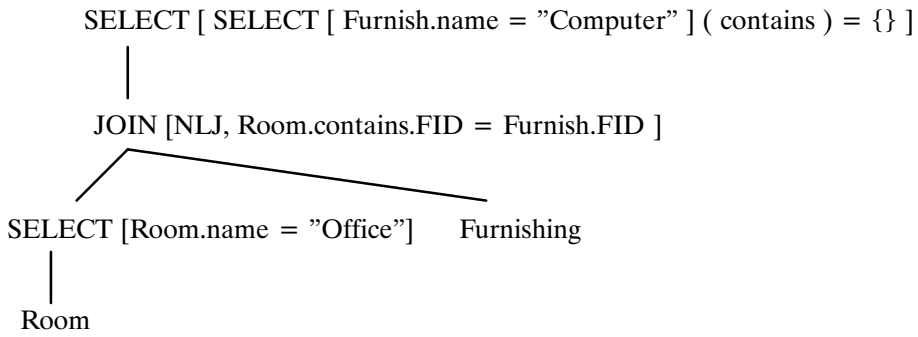
A query asking for offices without a computer, is formulated in COOL in the following way.

```
Q3 := select [ name = ”Office”
        and select [ name = ”Computer” ] ( contains ) != {} ]
      ( Room ) ;
```

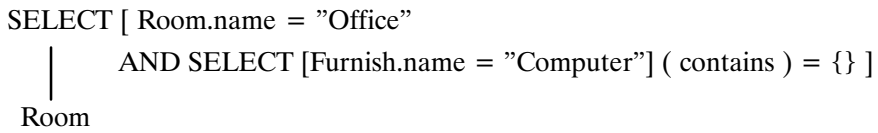
Assuming the physical design I (Figure 6.2), the translation to the physical schema results in a processing tree as given below.



Note that due to the use of the *contains* function (which is not materialized), a join operation is inserted. Because there is no partitioning or subtyping, no additional joins are necessary in this example. The '...' after the keyword *JOIN* indicate, that (up to now) no specific join method has bee:n selected. After optimization, the resulting execution plan looks like.:



Obviously it improves performance, to split the selection and move one part into the join, that is, to apply the *name* function first to *Room* objects, before following the set of links (*Room.contains*) to *Furnishings*, to check whether there is no *Computer* in that *Room.* Assuming an index on *name* of *rooms,* and physical links, this will be quite efficient, much more efficient, than in a flat relational approach. There the *Room–Furnishing* relationship is modeled by taking the primary key of *Rooms* into *Furnishings* (as a foreign key). However, if the physical design is the one given in Figure , query Q_3 results in



This execution plan does not even need any join operation, and can be executed within one (single scan) kernel operation.

The implementation of COOL on top of DASDBS has been completed (to the described level) only recently. Therefore, we can not yet provide any performance figures. The next step in our work, however, is to run such experiments and compare the relative performance of several possible physical designs.

Further, the DASDBS implementation will also be evaluated in comparison with to other implementations based on commercial platforms: Oracle, as a relational system that can emulate two-level nesting (via Oracle ”Clusters”), and Ontos, as one of the early commercial OODBMSs. First experiences with these two implementations are reported in [TS91], results concerning the DASDBS realization are given in [TRSB92, Lue92].

7. Mapping KRISYS to PRIMA

7.1 Mapping the KOBRA Model onto the Molecule Atom Data Model

In this section we now describe the mapping approach of KRISYS. We first introduce the complex-object data model - called *molecule atom data model* (or MAD model, for short) - onto which KOBRA is mapped. It was developed for supporting applications that require a suitable representation of complex objects, i.e., those whose inner structures (the components) are also objects of the DB [Hä88,HMMS87]. Due to space limitations, only basic features of the MAD model could be discussed in this section. More about the MAD model may be found in [Mi88, Schö89]. Then, we concentrate on how the knowledge structures of KOBRA are mapped to MAD.

7.1.1 A Short Overview of the Molecule Atom Data Model

The basic building blocks of the MAD model objects are *atoms*, which may be compared to tuples in the relational model. Atoms belong to a exactly one *atom type* (analogous to a relation) and consist of a set of attributes of various types. Besides the data types known from many implementations of the relational model, the MAD model allows for some additional data types, e.g. a TIME type and a HULL type, which represents n-dimensional rectangles. Furthermore complex types may be constructed by LIST and SET constructors. Each atom type must have one attribute of the type IDENTIFIER which contains a system-wide unique key for each atom.

Attributes of type REF_TO are used to link atom types to one another. The value of a REF_TO attribute is a duplicate-free list of IDENTIFIER values, all pointing to atoms of the same type. This link must be represented symmetrically, i.e., if an atom type A contains a REF_TO attribute pointing to atom B, then B must contain a REF_TO attribute pointing to A, too. Analogously, if the REF_TO attribute of an atom a of type A contains the IDENTIFIER value of atom b of type B, the corresponding REF_TO attribute of b must contain the IDENTIFIER value of a. The two REF_TO attribute values together form a link between a and b. Due to the symmetrical link concept, the atom types in a MAD schema form an undirected network, as well as the atoms of a MAD database do.

Based on these networks, *molecule types* and *molecules* may be defined by specifying a coherent subgraph of the atom type network and assigning a direction to each link type, such that the resulting directed graph has exactly one root (the so-called *root atom type*). Molecules are formed from the atom network by following the links among the atoms in the direction defined by the molecule type definition. In

contrast to the NF2 model [SS86] which expects NF2-objects to be defined in the DB-Schema, Molecule types and molecules are defined dynamically at query time.

The MAD model's query language is called MQL. An MQL retrieval operation has the following form

SELECT projection clause
FROM molecule type definition
WHERE restriction clause

The molecule type definitions contained in **FROM** clause may be very complex, e.g., they may define recursive molecule types. The **SELECT** clause is used to project only those parts of the molecule which are interesting. One can project all parts of a molecule (**ALL**), atoms of certain types or only some attributes of certain atom types. Furthermore, one can project atoms of a certain type only if they fulfil a qualification (qualified projection). The optional **WHERE** clause allows for a restriction of the molecule set specified in the **FROM** clause.

Besides the retrieval statement **SELECT** there are the manipulation operations **INSERT**, **DELETE** and **UPDATE** which also work on molecules. By means of the **INSERT** statement not only one molecule, but an arbitrary number of molecules may be inserted at a time. Furthermore, there are schema definition and manipulation statements.

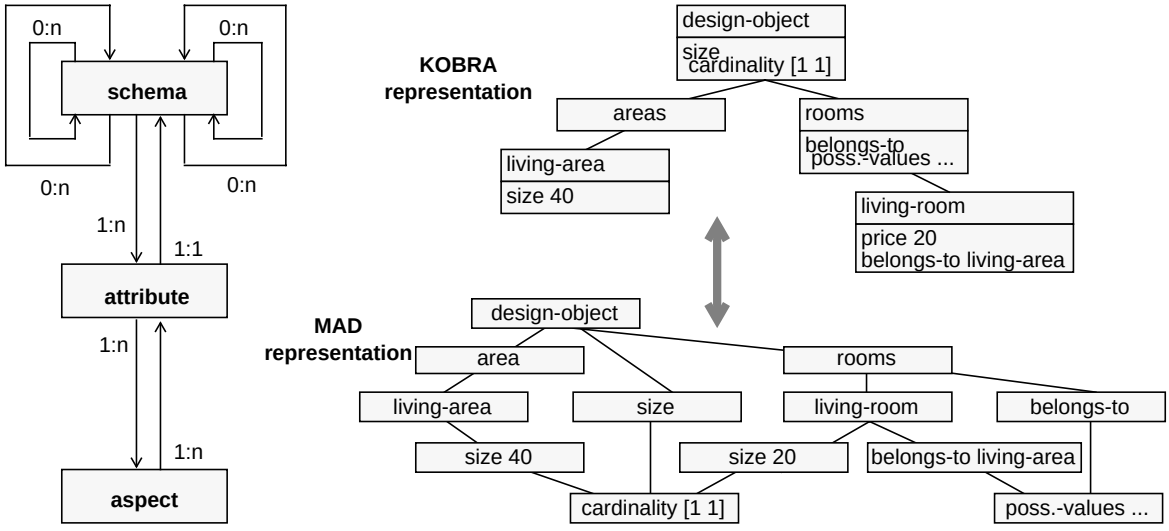
7.1.2 The Mapping Scheme of KRISYS

The efficiency of KRISYS functions directly depends on the chosen MAD schema for the representation of KOBRA structures together with the corresponding mapping mechanism. There are 2 different environments, in which KRISYS is used, with quite different requirements. During development of applications KRISYS is used as a modeling tool supporting an incremental modeling process and during application processing it is used as a runtime environment.

During application development

From the description of the development of knowledge-based applications given in chapter 4.1, it should be clear that for the construction or modeling phase, KRISYS has to provide a flexible mapping scheme especially appropriate for efficiently supporting functions that modify KB structures since object definitions and abstraction hierarchies are constantly being changed or extended. Hence, when mapping KOBRA structures to MAD, a separation between objects representing meta-information (e.g., classes) to be introduced in the DB schema of MAD and objects representing 'common' information (e.g., instances) to be included in the DB leads to an ineffective mapping because most KOBRA operations would provoke modifications on the MAD schema (e.g., to define, remove, expand, or shrink atom types, change attribute specifications, etc.). Thus, it is necessary to abstract from the role (i.e., class, instance, set, and so on) played by an object at the KOBRA level in order to be able to treat them all in the same manner at the MAD level only by means of operations on the DB. Since such roles are intrinsically related with the semantics of KOBRA, it is important to abstract from such semantics and consider KRISYS objects only from a structural point of view.

From such a viewpoint, schemas, attributes, and aspects are related to each other to compose the KOBRA objects, leading to a MAD schema (graphically illustrated in figure 7.1a) that contains three atom types (respectively 'schema', 'attribute', and 'aspect') connected via the references ('has_attributes' and 'has_aspects'). Here, aspect specifications are shared between several attributes preventing redundancy, especially in the case of inherited attributes (see figure 7.1b). We explicitly exploit the capability of the MAD model to handle network structures in a direct and non-redundant manner. (Such an adequate modeling is hard to achieve using data models that only allow for hierarchical structures, e.g., NF^2 models [SS86,Da86]). The attributes expressing abstraction relationships are not represented as ordinary 'attribute' but as recursive MAD references (e.g., 'has_subclasses', 'is_subclass_of', 'has_instances', etc.) between the atom type 'schema' (only partially shown in figure 7.1a). Since there is always a back reference for each MAD link, each of the organizational axes is represented by a pair of references.



a) MAD schema diagram

b) mapping process

Figure 7.1: Representation of KOBRA objects in MAD during application development

The mapping of KOBRA objects to MAD atoms is performed at the workstation side of KRISYS, whenever objects are stored into or discarded from its object buffer in order to allow a buffer representation embodying the semantics of the KOBRA model (figure 7.1b). Therefore, for the realization of KOBRA operations, the query language of MAD (MQL) is employed to fetch objects from the DB. For example, the definition of a new class in the middle of a generalization hierarchy, which requires the inheritance of its attributes, generates an MQL statement to select all relevant classes and instances from the DB. This task is accomplished by dynamically defining a recursive molecule starting with the root atom type 'schema' moving on to all subclasses (using the reference 'has_subclasses' in a recursive manner) and then to their instances (by exploiting the reference 'has_instances'). For each object reached, their attributes and aspects are also retrieved (via 'has_attributes' and 'has_aspects'). The selected part of the hierarchy is then modified in the object buffer, during which inheritance conflicts are solved, the inexistence of cycles or other ambiguities (e.g., an object having a same object as subclass and instance at the same time) in the hierarchy are checked, etc. At some time later on (probably at the end of a mod-

eling step), the modified hierarchy is then discarded from the buffer and propagated back to the DB, where it is updated via a single modify statement. This operation changes the recursive molecule in accordance to the given hierarchy by inserting atoms not yet stored and updating the corresponding connections. Note, however, that when introducing such modifications back into the server no changes need to be made in the schema information of MAD. Creations and deletions of KOBRA schemas (even classes, sets, or aggregates) are mapped to modifications on atoms of type 'schema'; manipulations of attributes (e.g., definitions, inheritance, etc.) provoke inserts, deletes, and updates on instances of the 'attribute' type; and changes on aspects (including their specialization) require only operations on 'aspect' atoms. Hence, the required flexibility for incrementally defining, correcting, and extending the KB during application development is well supported.

During application processing

The analyses of several knowledge based applications developed with KRISYS has shown 2 important aspects concerning the mapping scheme during application processing. Firstly, for application processing flexibility is in general no longer required. For this reason we can choose more specific mappings for this phase leading to significant gains in performance. Naturally, in order to exploit such specific mapping schemes, the MAD schema and the corresponding transformation process of KOBRA objects into MAD atoms must not be fixed to a certain mapping scheme (as during the modeling phase), but has to be tailored to the structures and processing characteristics of the application. The second important observation was, that the structures of KB's of the different applications as well as the mechanism used for representing procedural knowledge differ very much (figure 7.2 shown some criteria). Existing prototypes [Ni87,Ba87,Fi87] neglect the exploitation of application characteristics for optimization purposes presenting only fixed, predefined mapping schemes, which consider only a few characteristics and support only special kinds of queries well [WK90].

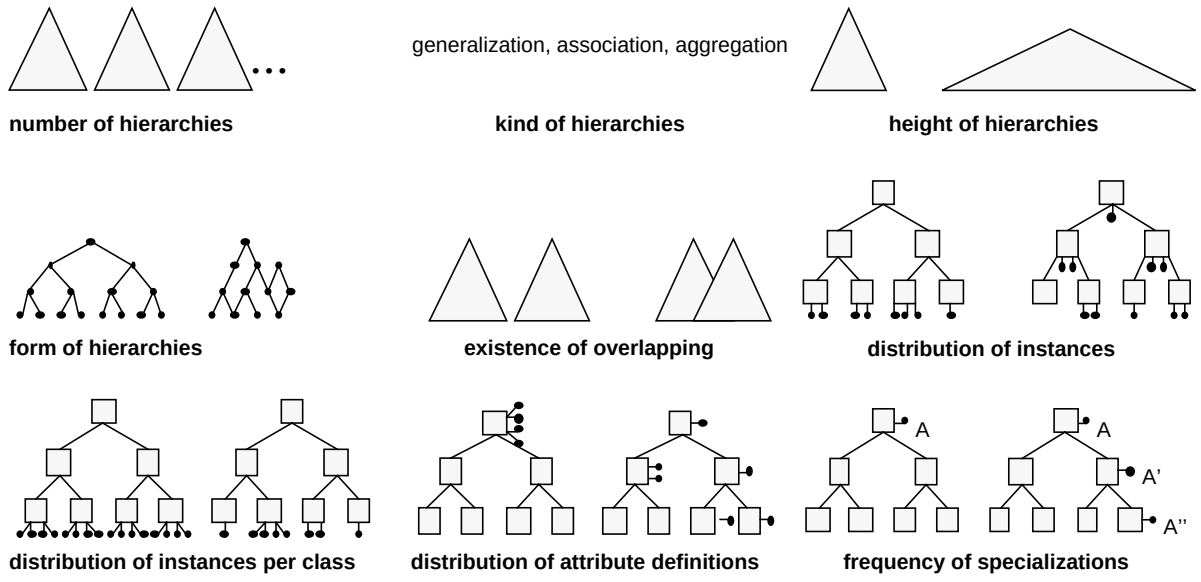


Figure 7.2: Some differences of KB structures of KRISYS applications

KRISYS, on the other hand, uses all such information to generate an application-oriented, and consequently very effective mapping scheme for each application. Our approach is to extend the development process of application by including an additional phase, called optimization, during which KRISYS specifies a new MAD schema tailored to the application at hand and automatically performs a DB transformation. The most appropriate DB schema for an application is determined by means of an analysis of the structures of the KB, that are completely specified and explicitly represented in KRISYS just before the optimization takes place [Ma90]. For example, the attributes of a KOBRA object, which are represented as separate atoms in the generic mapping, could all be represented as attributes of a single atom in the specific mapping, and classes could be used to define corresponding atom types. Once the new DB schema is generated, a compilation of the application is then performed, thereby also compiling the application queries encoded within methods, demons, and rules. In other words, the optimization phase represents the 'exit' of an interpretative (flexible but probably not so efficient) KB manipulation, which is necessary during KB construction, and the 'entry' into a compiled (efficient but not so flexible) KB manipulation, which is necessary during application operation.

Criteria for the generation of an application-oriented mapping

Although the generation of such new DB schema appears to be quite simple at a first glance, it involves the consideration of a lot of information and requires complex decisions to be made. Just as an example, not every class should generate an atom type because there might be classes that will never receive direct instances in a generalization hierarchy (e.g., the class design-objects in our design application). So, one could deduce a rule like 'every class possessing direct instances should have a corresponding atom type'. However, there are classes with direct instances, but presenting the same structure (i.e., attributes and aspect specifications) of a common superclass (e.g. the subclasses of rooms). They were (probably) defined only for organizational purposes. In this case, not the subclasses, but only this superclass should define an atom type, in which all instances would be inserted. One would now conclude that 'every class with different structure and direct instances should generate an atom type'. Nevertheless, even this conclusion is not always correct because there are a lot of cases in which grouping structural different classes has good reasons to be carried out. For example, when two such classes are used to define a set (e.g., the set representing the furnishings offered for someone contains many different kinds of furnishings e.g., beds, sofas or wardrobes, etc.), it might be important to put them together into one single atom type because KRISYS wants to exploit the corresponding membership stipulation or the attributes involved in the calculation of set properties to specify appropriate access paths. In this case, the only practical solution would be to create several different atom types and split the instances of the classes. One atom type includes the common attributes of these classes, on which the definition of the membership stipulation and the set properties is based (e.g., offered-to, price, width, height, length, etc.) and each of the other atom types having the individual attributes of each class (e.g., mattress for beds and no-of-seats for sofas, etc.). Naturally, in the case of queries searching for such objects, accesses to two different atom types would be necessary. However, this can be efficiently supported by using MAD references or even clustering mechanisms to keep both parts of each KOBRA object close together.

Furthermore processing characteristics of the applications are used to generate an appropriate mapping for an application. As in relational systems, the processing characteristics are used to generate an access path for attributes used frequently in queries. However, in the context of object-oriented query processing characteristics may be additionally used. In principle, two kinds of queries can be distinguished here: queries concerning all instances of a class (i.e., also the instances of the subclasses of the class) and those concerning only the direct instances of the class. The former can be supported very well by mapping the instances of the class and all subclasses to one atom type whereas the latter are most optimally supported by mapping each class to a separate atom type. However, this criteria may be in conflict with the criteria concerning the structure of a hierarchy (e.g., since the structure of the subclasses of a class differ very much, but queries concerning all instances of the class occur frequently). Furthermore, classes may exist, which are accessed via both kinds of queries. For those classes must be decided which of the two mapping schemes should be used.

In figure 7.3 we show the mapping of a classification hierarchy of our design environment based on the criteria explained above.

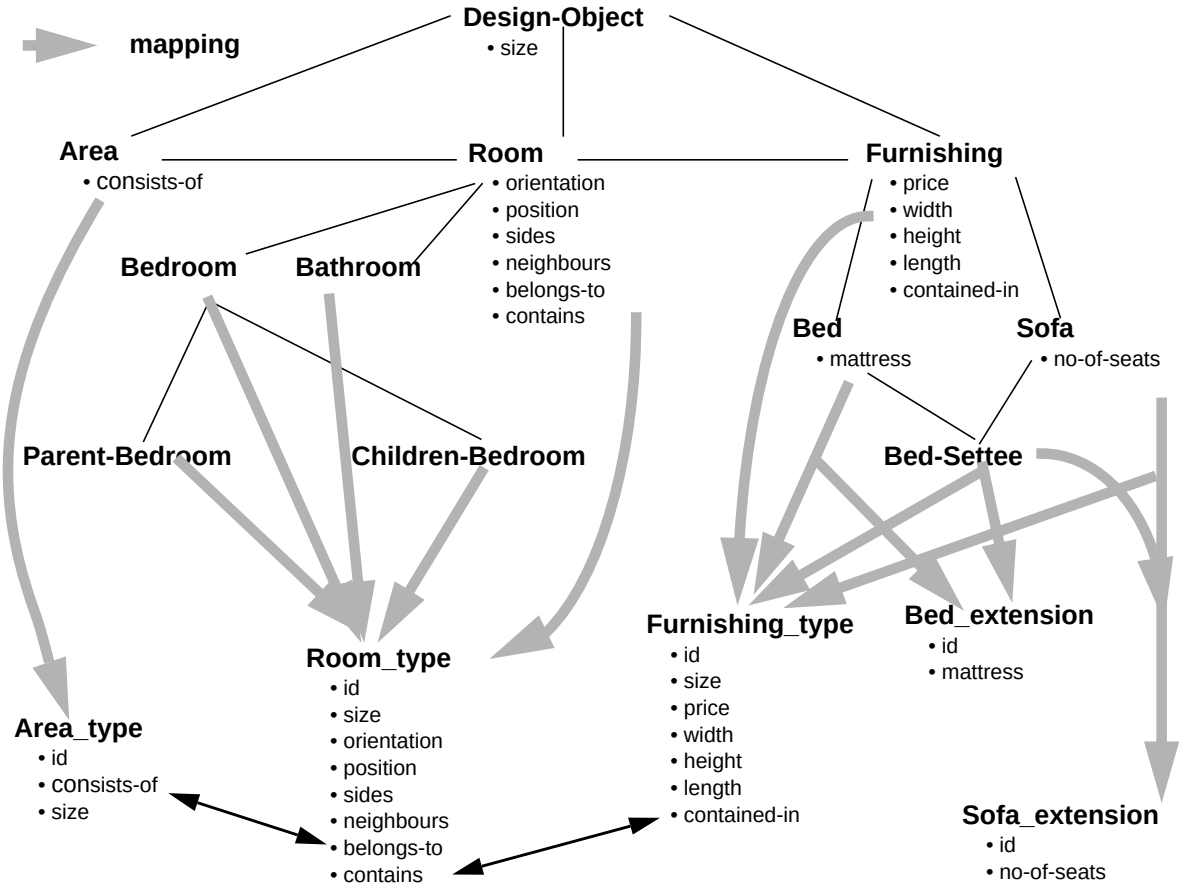


Figure 7.3 : Mapping of KOBRA objects to MAD during application processing

The mapping layer of KRISYS

From the short discussion above, one can conclude that there is no fixed procedure to be followed in order to derive the best MAD schema for an application, but only some heuristics that have to be carefully considered. Hence, the use of rule-based reasoning mechanisms seems to be very appropriate here since they provide the required expressiveness and flexibility to represent and interpret such heuristics. So, during the optimization phase, KRISYS exploits the rule-based mechanisms provided by the KOBRA model to generate a tailored mapping scheme. For this purpose, it interacts with the knowledge engineer whenever necessary in order to get additional, relevant information for making its decisions (e.g., the number of expected instances per class, elements per set, the relevance of membership stipulations for access purposes, etc.). In the case of an erroneous estimation, the optimization phase would then be repeated at some time in the future in order to generate a new, more appropriate MAD scheme. KRISYS will then reconsider the increasing costs of this new transformation since several objects might now be stored in the KB so that the same issues used for optimization of DB design can be applied here.

This task is performed by a component, called *MAD-Schema-Generator* (see fig. 7.4). Beside the MAD scheme, it also generates information about the mapping of classes and sets to MAD atom types, the socalled *mapping-information,* which is used by the *mapping component*, in order to generate an MQL-Query, when read or write operations occur. In the case of operations provoking schema-evolutions the *transformation-component* updates the mapping information concerned.

By means of this framework the application is not affected at all, when a transformation of the MAD schema is performed or repeated, since it still sees only the unchanged KB view provided by the KOBRA model. This independence is in turn exploited as basis for generating various, suitable mapping schemes, thereby providing a kind of adaptability of our system to support the 'contradicting' requirements of flexibility and efficiency.

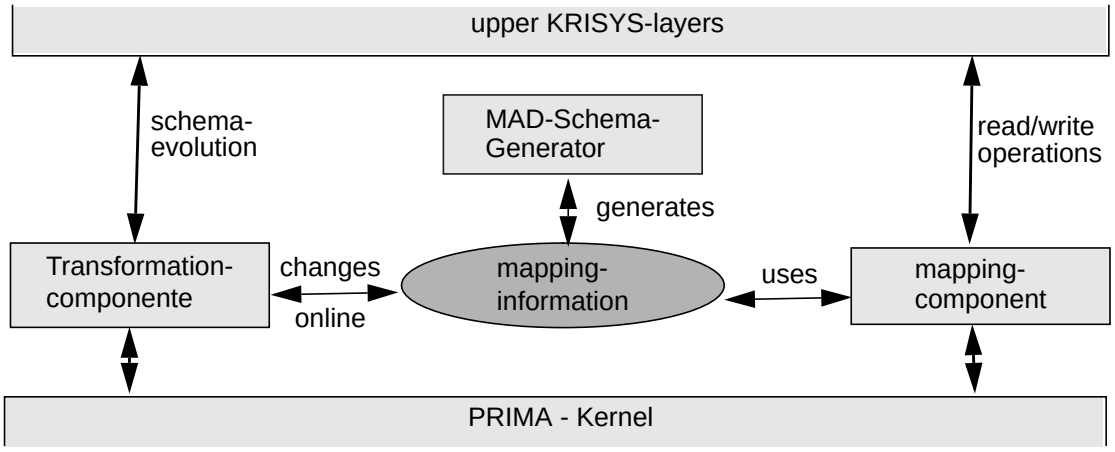


Figure 7.4. Mapping Layer of KRISYS

7.2 The Processing Model of KRISYS

In the previous section we have described the mapping of KOBRA object structures onto the MAD model, but we have not yet explained, how KRISYS operations (e.g., KOALA statements) are supported by the PRIMA kernel. Since PRIMA provides a powerful data manipulation language (MQL) at it's interface, one might expect an approach to be feasible, where each KOALA statement is evaluated by transforming it into a (sequence of) equivalent MQL statements to be executed by PRIMA. At a closer look, however, one realizes, that this is not an adequate solution, mainly because the approach does not match the processing characteristics resulting from the hardware environment for which KRISYS was conceived: The architecture of KRISYS should fit into a workstation/server environment with decentralized and autonomous processors.

In the following we will describe the impact of this processing environment on the processing model of KRISYS and sketch the approach we have actually taken.

7.2.1 Implications of a Workstation/Server Environment

In a workstation/server environment, applications run on powerful workstations exploiting local processing capabilities, main memory, and (probably) disk space, which are usually dedicated to single users and have access to a central server component providing and controlling the use of centralized information (i.e., the OB). In such an environment, the coupling of the workstation and server components becomes a performance-critical issue. Minimizing communication traffic and OB access emerges as the primary goal, which can only be achieved by a loose coupling approach exploiting the high degree of locality of reference of the application on the workstation side.

One step towards this goal is to integrate a main memory buffer (called working-memory, WM, in KRISYS) into the workstation component, providing a facility for representing relevant OB contents close to the application and allowing the accumulation of manipulations, thereby reducing communication with the server component to a reasonable amount [LM89]. This is, however, not the only step to be taken. In the design of the overall architecture, one has to be extremely cautious with the choice of the server component interface, because it determines the server functionality and therefore also the amount of processing actually delegated to it.

Looking at the main architectural components of KRISYS, one easily recognizes a kind of semantic gap between KOBRA, the object model of KRISYS, and the MAD model (see sections 4.1 and 7.1). Several KOBRA concepts, like the abstraction concepts of association and generalization as well as methods, demons and rules, are not known in MAD. The question arises, where to put the borderline between server and workstation w.r.t. these concepts. Because especially the operational concepts (i.e., methods, demons, and rules) may contain significant parts of the application, it is not desirable to include them in the server functionality [De91,Ma91]. This would actually shift a large amount of application - oriented processing from the workstation to the (then very powerful) server, which would then be hopelessly overloaded, thereby neglecting the idea of workstation-oriented processing locality and increasing

the overall communication overhead. Consequently, we have chosen the MAD model as the interface of our server component and have placed KOBRA and KOALA at the workstation (see fig. 7.5).

Additionally, we have decided to represent contents of the OB at the workstation in a form more appropriate for the realization of the KOBRA/KOALA functionality, i.e., as objects of the KOBRA model [La91]. When information is transferred from the server to the workstation, a transformation according to the mapping scheme described in section 7.1 takes place, so that the objects transferred in the form of MAD molecules obtain KOBRA semantics as they are placed in the WM (see fig. 7.5). This has the advantage, that the functionality of KOBRA and KOALA may be realized on the basis of a semantically enhanced data structure, where, for example, object structures and abstraction hierarchies are already “materialized”. Therefore, the semantic gap between KOBRA and MAD has to be bridged only once (i.e., as objects are transferred between server and workstation) and not each time an operation on an object is performed. Otherwise, the KRISYS functionality (i.e. KOALA queries, method execution, etc.) would also have to be mapped directly to the MAD model, frequently requiring complex transformations and tedious interpretation of results produced at a lower semantic level.

Summarizing, the KRISYS architectural approach for workstation/server environments is strongly motivated by the primary goal to leave the main parts of application processing at the workstation, while the server component supports application-independent data management tasks at it’s interface, the MAD model.

7.2.2 Query Evaluation in KRISYS

In the following we will only give a brief sketch of the query evaluation process in order to point out the most important issues (see [Hä91, daRo92] for details). A thorough discussion would certainly exceed the space limitations of this paper.

As one can see from the above description, the decision to support processing of KOALA queries at the workstation is influenced by two points. First of all, there is the need to strongly exploit the locality of reference during the processing of the application (which may be a long term activity). Therefore, query processing has to repeatedly exploit the contents of the WM. Secondly, query evaluation is strongly interwoven with other operational concepts like method execution, and the evaluation of rules and de-

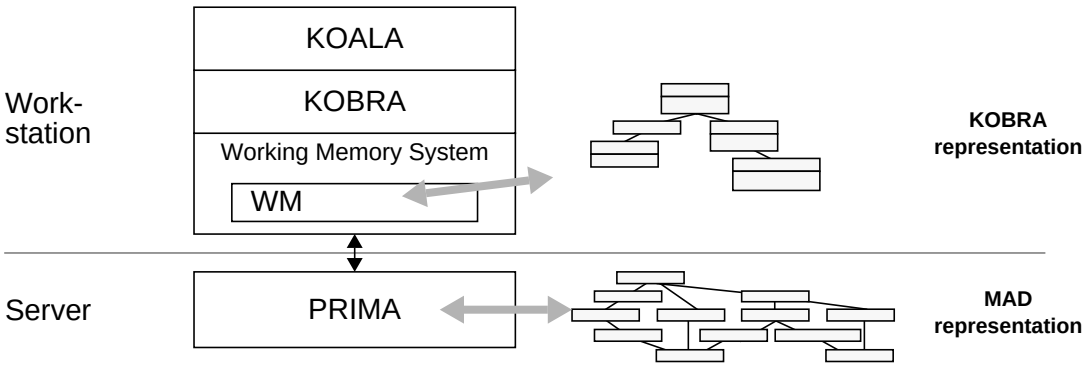


Figure 7.5: A Refined View of the KRISYS Architecture

mons, as well as with the semantics of the abstraction concepts, which are realized at the workstation for the reasons we have explained above.

From a general point of view, query evaluation at the workstation follows the approach usually taken in DBS. We have defined an algebra for our query language, which resembles the basis for efficient, set-oriented processing of KOALA queries. Rule-based optimization techniques are employed to transform a query into an efficient execution plan. The plan operators for workstation-based query evaluation work on specific access structures containing sets of objects. Since applications of increasing complexity (e.g., engineering systems) may require large amounts of objects to be kept permanently at the workstation, main-memory indices may be installed and utilized for query processing.

Until now, we have not talked about the interaction of workstation and server. How are objects requested to be transferred from the server into the WM (and vice versa), and in what granules does the transfer occur? In order to minimize the interaction between the components, we can formulate two basic requirements. First of all, the transfer should be set-oriented, meaning that the unit of transfer should be a set of objects, and not single objects, which would increase the communication overhead. Secondly, an associative specification of the objects to be transferred (i.e., in terms of a declarative query) should be supported. Otherwise, objects could only be qualified via their identifiers, imposing a more or less navigational style of processing at the workstation, which is far from efficient for the evaluation of declarative KOALA queries. Also, the possibilities of the server component to pre-select objects relevant for application processing would be completely neglected.

It should have become clear from previous parts of this section, that a complete delegation of a KOALA query to the server is in general not desirable and (usually) not possible. Therefore, we support the delegation of subqueries (i.e., complete subtrees of the algebra graph for our KOALA queries) to the PRIMA component during query evaluation at the workstation. Special plan operators in our execution plan are used to indicate, that parts of the query should be transformed into an equivalent MQL query using the mapping information. Due to the semantic gap, this is usually not possible for all parts of a query, but the optimizer can always construct an algebra graph, which contains select-operations as it's leaves that can be delegated. Consider, for example, the KOALA query

```
(ASK   ((?X)(?Y))
       (AND  (IS-INSTANCE ?X rooms *)
             (EQUAL South (SLOTVALUE orintation ?X))
             (IS-INSTANCE ?Y furnishings *)
             (> (SLOTVALUE price ?Y) 1000)
             (MESSAGE is-suitable-for ?Y ?X)))
```

selecting all furnishings costing more than 1000 $, which are suitable for rooms located at the south side of the house. This query, which contains a method activation (indicated by the MESSAGE predicate) for determining suitability, can be transformed into the algebra graph shown in figure 7.6 where both select operations may easily be transformed into MQL statements in order to delegate their execution. The join operation has to evaluated at the workstation. ,

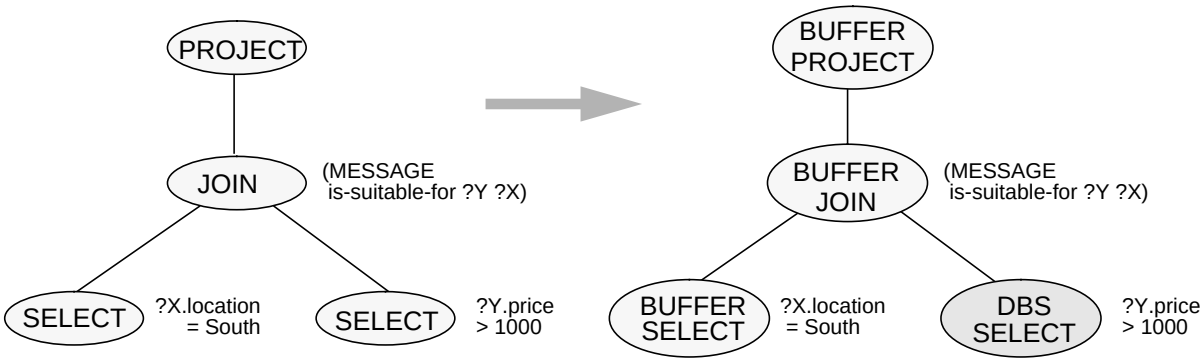


Figure 7.6: Delegation of Subqueries

In general, however, not all the leaves of an algebra graph are delegated to the server. Since queries are frequently executed during application processing, we have to take the contents of the WM into account in order to preserve the processing locality of the application. Therefore, we delegate only those subqueries that require information not present at the workstation for their evaluation. In order to evaluate the above query, it may, for example, only be required to delegate the select operation involving furnishings (see figure 7.6), because the required rooms are already present in the WM. In order to decide which subqueries have to be delegated, we keep a declarative description of our WM contents, which is maintained by a component called Context Manager (CM). Each time a subquery is delegated and the results of this subquery are brought into the buffer, the CM treats this result as a so-called context and updates the description. For example, after the above query, the fact that all furnishings with price > 1000 are contained in the WM would be registered. The CM is not only responsible for maintaining the WM description, but can also perform subsumption tests in order to interrelate contexts and determine, if sets of objects characterized by subqueries are contained in the WM. These subsumption tests, which also take into account semantic information (e.g. the subclass hierarchies), can then be exploited by the query optimizer in order to generate the right execution plan.

A detailed elaboration of the CM, which has been developed in the scope of a master thesis [St92], is clearly beyond the scope of this paper. Therefore a lot of questions that arise from the above sketch with respect to updates, check-in of objects into the server, partial subsumption of contexts, etc., have to remain unanswered here. We nevertheless hope that we have given an impression of the query evaluation approach taken in KRISYS.

8. Comparison of the mapping approaches

At a first glance the mapping of COCOON and KRISYS onto secondary storage looks quite similar. Both systems follow the so-called database system kernel approach [HR85] using a database management system kernel (DBMS kernel) - DASDBS in the case of COCOON and PRIMA in the case of KRISYS - in order to manage complex structured data on secondary storage. Following this approach, COCOON as well as KRISYS are concerned with two identical questions: how to map the object model to the model

of the underlying DBMS kernel and how to use the functionality of the DBMS kernel during query processing, i.e., the kind of interaction between the DBMS kernel and the upper components of the system.

However, a closer look to the two approches described in the previous sections shows some important differences concerning the functionality of the DBMS kernel as well as the mapping of the object model and the interaction between the DBMS kernel and the upper components of the two systems.

Functionality of the DBMS kernel

DASDBS, the DBMS kernel of COCOON, is a complex object storage system according to the NF^2 model. It supports general NF^2 structures, that is, it allows for the storage of hierarchical clustered data with an arbitrary level of nesting, supports hierarchical access structures, and for example the opportunity to define nested join indizes. The structure of the complex objects is defined statically in the database schema. Network and recursive structures cannot be modeled directly at this level. DASDBS offers a powerful descriptive, set-oriented, algebraic interface with efficient operations on complex objects. All kernel operations are performed in a single scan through the data.

On the other hand, PRIMA, the DBMS kernel of KRISYS supports the MAD model, which allows the definition of complex objects of an arbitary structure (even network and recursive structures), called molecules, from simple structured basis-objects, called atoms, which are building a network by means of structural connections. Molecules are defined dynamically at query time. At its interface PRIMA offers a powerful, descriptive and set-oriented Query Language (MQL) which enables the selection of arbitrary complex and even recursive molecules built from atoms of several types. Furthermore, MQL even supports the qualified projection of atoms.

Interaction between the DBMS kernel and the upper layers

In order to perform COOL queries a strong interaction is possible between COCOON and DASDBS. COOL is very similar to the nested algebra of DASDBS, which enables an efficient transformation and optimization, resulting in query processing in an integrated manner.

KRISYS follows a totally different approach with respect to the database system kernel architecture. In order to support processing in a workstation/server environment, the DBMS kernel which is running on the server is only loosly coupled with the upper components of KRISYS running on a workstation. KRISYS maintains some kind of application buffer, the working-memory, which temporarily stores objects, in order to use the locality of the application on the workstation. Based on a declarative description of the WM contents, KRISYS realizes a hybride processing scheme, which allows parts of a query to be processed at the workstation, thereby exploiting workstation locality, while others may be delegated to the server for execution.

Mapping of the object model

COCOON strictly separates OB schema and instance information. Furthermore is was developed as an object management system. For these reasons, COCOON can easily provide a simple default mapping.

Nevertheless, there are many possibilities of mapping the objects to the NF^2 model. This results in being superior, due to flexible database layouts, whose efficiency can be tailored to operations performed on the objects as well as the kind of relationship between the objects. For this reason, COCOON uses a physical design tool in order to choose a most efficient mapping w.r.t. the overall transaction load faced.

KRISYS, on the other hand, was designed as a modeling tool as well as a runtime environment. Therefore no distinction between schema information and instance information is made in KRISYS. During the application development phase KRISYS supports a flexible, but less efficient mapping to PRIMA, since schema modifications occur very often and a lot of information needed to define a adequate mapping is not known (e.g., the type of the attribute or its cardinality). Similar to COCOON and for the same reasons it uses an expert system in order to generate an appropriate mapping for application processing after the completion of the modeling phase.

9. Conclusions and Outlook

In this paper we have presented an overview and a synoptic comparison of two OODBMSs, COCOON and KRISYS, which have been developed at the ETH Zürich and at the University of Kaiserslautern, respectively. The architecture of both systems follows the database kernel system approach, where the functionality of the object model and the language is supported by a database kernel providing common data management tasks.

A closer look at the architectural layers of COCOON and KRISYS reveals not only commonalities, but also a number of important differences. Besides the presence or absence of some modeling ore language constructs in one or the other system, our comparison has also revealed more fundamental differences, which are to a large extent provoked by differing key assumptions under which the systems have been developed.

First of all, one has to look at the basic functionality provided by each system. While COCOON was developed as a pure object management system with emphasis on efficiency, update integrity and schema evolution, one of the main goals in the design of KRISYS was to additionally provide the functionality of a modeling tool, i.e., system support for incremental OB and application development. The influences of these different preassumptions can be located at all layers of the system. COCOON, on one hand, strictly separates OB schema and instance information and follows an evolutionary approach, extending efficient language and processing concepts of relational-style systems. KRISYS, on the other hand, integrates schema information into the OB, supports a ‘dynamic’ behavior of the abstraction concepts, and provides access to meta-information in it’s query language in order to realize the required functionality. This basic requirement also influences the approaches for mapping object structures onto the underlying DBMS kernels. While COCOON can provide a default mapping scheme which is relatively close to an optimized one, KRISYS uses a default mapping tailored towards the application development phase, which is far less efficient than the optimized mapping, but easily allows schema modifications.

The second key assumption refers to the hardware environment the systems should match. COCOON more or less assumes a centralized environment, while KRISYS fits into the general processing scheme of workstation/server environments. Therefore, the distinct architectural layers may be integrated more closely in COCOON w.r.t. their interaction and the overall functionality than in KRISYS. Here, the system consists (from a hardware point of view) of two merely independent, loosely coupled components, namely the workstation component, providing the object model and query language functionality, and the server, where the DBMS kernel is located. Consequently, the processing models of both systems differ significantly. In COCOON, query processing is performed in an 'integrated' way in one system component, while KRISYS supports query evaluation at the workstation *and* the server, in order to allow a 'two-staged' execution strategy, which exploits processing locality at the workstation. In the same way, the two DBMS kernel systems differ in the complexity of tasks to be performed in the overall system. In COCOON, the internal DASDBS kernel interface supports complex operations on one NF2 relation, allowing efficient evaluation in a single scan through the data, whereas some complex operations have to be performed in the high-level query processor in order to evaluate COOL queries. However, the PRIMA system provides full-fledged DML functionality, thereby also allowing the delegation of complex subqueries from the workstation to the server.

Because of their significant effects revealed by our comparison, we can conclude that the above mentioned key assumptions can be seen as important points of choice in designing an OODBMS w.r.t. their impact on the functionality to be provided by the distinct system components.

In both projects, future activities will be directed towards the refinement of the above presented concepts, which will also involve performance evaluations. In KRISYS, especially the processing model and the interaction between workstation and server during query processing offers a large field for research activities. For example, the exploitation of parallelism in this framework seems to be promising, and is currently investigated in the project. In COCOON, the implementation of efficient update and schema evolution operations is currently in progress. Performance experiments of COCOON realizations on top of DASDBS as explained here and on top of Oracle are being carried out. The results of these activities will certainly offer more potential for another in depth comparison in the future.

Literature

[ACO85] Albano, A. and Cardelli, L. and Orsini, R., Galileo: A strongly-typed, interactive conceptual language, ACM Transactions on Database Systems, 10(2), p.230, June 1985.

[Ba87] Banerjee, J., et al.: Data Model Issues for Object-Oriented Applications, in: ACM Transactions on Office Information Systems, Vol. 5, No. 1, 1987, pp. 3-26.

[BD87] Brauburger, D. and P. Deusser, Entwurf und Implementierung eines Kostenschaetzers für Kern- (CRM) Operationen sowie eines physischen Datenbank-DESIGNER's für DASDBS, Diplomarbeit,Technische Hochschule Darmstadt, Fachbereich Informatik, Darmstadt, 1989.

[Bee89] Beeri, C. : Formal Models for Object Oriented Databases (Invited Paper), Proc. First Int'l. Conf. on Deductive and Object– Oriented Databases, Kyoto, Japan, December 4–6, 1989. Revised version appeared in Data & Knowledge Engineering,Vol.5, North–Holland.

[Ber90] Bertino, E., Optimization of Queries Using Nested Indices, Lecture Notes in Computer Science, vol. 416, p. 44, Springer Verlag, Venice, Italy, March 1990.

[BFL83] Brachman, R.J., Fikes, R.E., Levesque, H.: KRYPTON: A Functional Approach to Knowledge Representation, in: Computer, Vol. 16, No. 10, Oct. 1983, pp. 67-73.

[BMW84] Borgida, A., Mylopoulos, J., Wong, H.K.T.: Generalization/Specialization as a Basis for Software Specification, in: On Conceptual Modelling (Perspectives from Artificial Intelligence, Databases, and Programming Languages), Topics in Information Systems, (eds.: Brodie, M.L., Mylopoulos, J., Schmidt, J.W.), Springer-Verlag, New York, 1984, pp. 87-114.

[BS83] Bobrow, D.G., Stefik, M.: The LOOPS Manual, Xerox PARC, Palo Alto, CA, 1983

[Da84] Date, C.J.: Some Principles of Good Language Design, in: ACM SIGMOD Record, Vol. 14, No. 3, Nov. 1984, pp. 1-7.

[Da86] Dadam, P., et al.: A DBMS Prototype to Support Extended NF^2-Relations: An Integrated View on Flat Tables and Hierarchies, in: Proc. ACM SIGMOD Conf., Washington, D.C., 1986, pp. 356-367.

[daRo92] da Rocha, R.P.: Transformation and Rewrite in the Query Processing System of the KBMS KRISYS (in portuguese), Master Thesis, CPGCC, UFRGS, Porto Alegre, Brasil, May1992.

[De91] Deßloch, S.: Handling Integrity in a KBMS Architecture for Workstation/Server Environments, in: Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", Kaiserslautern, März 1991, Hrsg. H.-J. Appelrath, Informatik-Fachberichte 270, Springer-Verlag, S.89-108.

[DHLM92] Deßloch,S., Härder, T., Leick F.-J., Mattos N.: KRISYS - A KBMS Supporting the Development and Processing of Advanced Applications, in:Objektbanken für Experten (Eds. R.Bayer, T.Härder, P. Lockemann), Informatik aktuell, Springer Verlag, Heidelberg,1992.

[DLM90] Deßloch, S., Leick, F.-J., Mattos, N.M.: A State-oriented Approach to the Specification of Rules and Queries in KBMS, ZRI-Report 4/90, University of Kaiserslautern, July 1990.

[DMB+87] Dayal, U., Manola, F., Buchmann, A., Chakravarthy, U., Goldhirsch, D., Heiler, S., Orenstein, J., Rosenthal A.: Simplifying Complex Objects: The {PROBE} Approach to Modeling and Querying Theme, In. H.–J. Schek, G. Schlageter, editors, Proc. GI Conf. on Database Systems for Office, Engineering and Scientific Applications, p.17, Darmstadt, Germany, April 1987, IFB 136, Springer Verlag,Heidelberg.

[DS86] Dayal, U., Smith, J.M.: PROBE: A Knowledge-Oriented Database Management System, in: [BM86], pp. 227-257.

[Fi87] Fishman, D.H., et al.: IRIS: An Object-Oriented Database Management System, in: ACM Transactions on Office Information Systems, Vol. 5, No. 1, 1987, pp. 48-69.

[FK85] Fikes, R., Kehler, T.: The Role of Frame-based Representation in Reasoning, in: Communications of the ACM, Vol. 28, No. 9, Sept. 1985, pp. 904-920.

[FWA85] Fox, M., Wright, J., Adam, D.: Experience with SRL: an Analysis of a Frame-based Knowledge Representation, Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh 1985.

[GD87] Graefe, G., DeWitt, D.J.: The EXODUS Optimizer Generator, Proc. ACM SIGMOD Conf., p. 160, San Francisco, CA, May 1987.

[GMN84] Gallaire, H., Minker, J., Nicolas, J.-M.: Logic and Databases: A Deductive Approach, in: ACM Computing Surveys, Vol. 16, No. 2, June 1984, pp. 153-185.

[Goe92] Goehring, A.: Adaptiver physischer DBEntwurf für objektorientierte Datenbanksysteme, Diplomarbeit, Zürich, 1992.

[Gro91] Gross, R.: Physischer Datenbankentwurf für objekt–orientierte Datenbanksysteme, Diplomarbeit, ETH–Zürich, Departement Informatik, Zürich, 1991.

[Hä88] Härder, T. (ed.): The PRIMA Project Design and Implementation of a Non-Standard Database System, SFB 124 Research Report No. 26/88, University of Kaiserslautern, Kaiserslautern, 1988.

[Hä91] Hänsel, E.: Die Anfragenverarbeitung im Wissensbankverwaltungssystem KRISYS, Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1991.

[HMMS87] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. 13th VLDB Conf., Brighton, UK, 1987, pp. 433-442.

[Hof92] Hofmann, J.: Evaluierung eines Anfrageoptimierers, Semesterarbeit, ETH–Zürich, Departement Informatik, Zürich, 1992.

[HR85] Härder, T., Reuter, A.: Architektur von Datenbanksystemen für Non-Standard-Anwendungen, in: Proc. GI Conf. on Database Systems for Office, Engineering and Scientific Applications, p.253-286, Karlsruhe, Germany, März85, IFB 94, Springer Verlag,Heidelberg.

[Ki89] Kim, W.: A Model of Queries for Object-Oriented Databases, in: Proc. of the 15th VLDB Conf., Amsterdam, August 1989, pp. 423-432.

[KM90a] Kemper, A. and Moerkotte, G., Advanced Query Processing in Object Bases Using Access Support Relations, Proc. Int'l. Conf. on Very Large Data Bases, p. 290, Brisbane, Australia, 1990.

[KM90b] Kemper, A. and Moerkotte, G., Access Support in Object Bases, Proc. ACM SIGMOD Conf., p. 364, Atlantic City, NJ, May 1990.

[Kr89] The KBMS Prototype KRISYS - User Manual, Version 2.3, Kaiserslautern, West Germany, 1989.

[Lae91] Laes, T.: Generierung und Evaluierung eines Anfrageoptimierers, Diplomarbeit, ETH–Zürich, Departement Informatik, Zürich, 1991.

[La91] Langkafel, D.: Eine Komponente zur graphenorientierten Verwaltung von Wissensbankausschnitten, Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1991.

[LB86] Levesque, H.J., Brachman, R.J.: Knowledge Level Interfaces to Information Systems, in: [BM86], pp.13-34.

[LM89] Leick, F.-J., Mattos, N.M.: A Framework for an Efficient Processing of Knowledge Bases on Secondary Storage, in: Proc. of the Fourth Brazilian Symposium on Data Bases, Campinas - Brazil, April 1989.

[LS92] Laasch, C., Scholl, M.H.: Generic Update Operations Keeping Object–Oriented Databases Consistent, Proc. GI Workshop Information Systems and Artificial Intelligence, Ulm, Germany, February 1992, IFB 303, Springer Verlag, Heidelberg.

[LS93] Laasch, C., Scholl, M.H.: Deterministic Semantics of Set-Oriented Update Sequences, in , Proc. IEEE Conf. on Data Engineering, Vienna, Austria, April 1993.

[Lue92] Luethi, J.M., Evaluierung verschiedener Joinalgorithmen auf einem NF2 DBS-Kern, Semesterarbeit, ETH-Zürich, Departement Informatik, Zürich, 1992.

[LV91] Lanzelotte, R., Valduriez, P., Ziane, M., Cheiney, J.J.: Optimization of Nonrecursive Queries in OODBs, Proc. Conf. on Deductive and Object–oriented Databases, Munich, Germany, December 1991.

[Ma88a] Mattos, N.M.: Abstraction Concepts: the Basis for Data and Knowledge Modeling, in: 7th Int. Conf. on Entity-Relationship Approach, Rom, Italy, Nov. 1988, pp. 331-350.

[Ma90] Mattos, N.: An Approach to DBS-based Knowledge Management (invited talk), in: Proc. 1st Workshop "Information Systems and Artificial Intelligence", Ulm - West Germany, March 1990, pp. 127-152.

[Ma91] Mattos, N.M.: An Approach to Knowledge Base Management, Lecture Notes in Artificial Intelligence (subseries of Lecture Notes in Computer Science), Vol. 513, Springer, Berlin, 1991.

[Ma91a] Mattos, N.M.: KRISYS - a KBMS Supporting Development and Processing of Knowledge-based Applications in Workstation/Server Environments, ZRI-Bericht 5/91, Universität Kaiserslautern, submitted for publication.

[May92] Mayer, O.: Abbildung und Integration von COOL–DDL auf NF2, Semesterarbeit, ETH–Zürich, Departement Informatik, Zürich, 1992.

[Mey88] Meyer, B., Object-Oriented Software Construction, International Series in Computer Science. Prentice Hall, Englewood Cliffs, 1988.

[Mi88] Mitschang, B.: Towards a Unified View of Design Data and Knowledge Representation, in: Proc. of the 2nd Int. Conf. on Expert Database Systems, Tysons Corner, Virginia, April 1988, pp. 33-49.

[MM89] Mattos, N.M., Michels, M.: Modeling with KRISYS: the Design Process of DB Applications Reviewed, in: Proc. the 8th Int. Conf. on Entity-Relationship Approach, Toronto - Canada, Oct. 1989, pp. 159-173.

[MMM92] Mattos, N.M., Meyer-Wegener, K., Mitschang, B.: Grand Tour of Concepts for Object-Orientation from a Database Point of View, to appear in: Data and Knowledge Engineering.

[Ni87] Nixon, B., et al.: Implementation of a Compiler for a Semantic Data Model: Experiences with Taxis, in: Proc. of ACM SIGMOD Conf., San Francisco, 1987, pp. 118-13.

[OL88] Ono, K. and Lohman, G.M., Extensible Enumeration of Feasible Joins for Relational Query Optimization, IBM Research Report, vol. RJ 6625 (63936), San Jose, CA, December 1988.

[Pau88] Paul, H.-B., DAS Database Kernel System for Standard and Non-Standard Applications -Architecture, Implementation, Application-, Ph.D. Thesis, Dept. of Computer Science, TU Darmstadt, 1988.. (in German)

[PB89] Pathak, G. and J.A. Blakeley, Query Optimization in Object-Oriented Databases, in Proceedings of the Workshop on Database Query Optimization, ed. G. Graefe, p. 115, Oregon Graduate Center Computer Science Technical Report 89-005, Beaverton, OR., May 1989.

[PSS+87] Paul, H.B., H.J. Schek, M.H. Scholl, G. Weikum, and U. Deppisch, Architecture and Implementation of the Darmstadt Database Kernel System, Proc. ACM SIGMOD Conf., p. 196, San Francisco, CA, May 1987.

[RHMD87] Rosenthal, A., Heiler, S., Manola, F., Dayal, U.: Query Facilities for Part Hierarchies: Graph Traversal, Spatial Data, and Knowledge-Based Detail Supression, Research Report, CCA, Cambridge, MA, 1987

[RRS92] Rosenthal, A., Rich, C., Scholl, M.H.: Reducing Duplicate Work in Relational Join(s): A Unified Approach, Submitted for publication, also ETH Zürich, Dept. of Computer Science, Technical Report NR: 172, 1992.

[RS93] Rich, C. Scholl, M.H., Query Optimization in an OODBMS, in GI Conf. on Database Systems for Office, Engineering and Scientific Applications, Informatik aktuell, Springer Verlag, Braunschweig, Germany, March 1993.

[SC90] Shekita, E.J. and Carey, M.J., A Performance Evaluation of Pointer-Based Joins, Proc. ACM SIGMOD Conf., p. 300, Atlantic City, NJ, May 1990.

[Sch86] Scholl, M.: Theoretical Foundation of Algebraic Optimization Utilizing Unnormalized Relations, Proc. ICDT, p. 380, Rome, Italy, September 1986

[Sch88] Scholl, M.H.: The Nested Relational Model – Efficient Support for a Relational Database Interface, Ph.D. Thesis, Dept. of Computer Science, TU Darmstadt, 1988. (in German)

[Sch89] Schöning, H.: Integrating Complex Objects and Recursion, in: Proc. 1st Int. Conference on Deductive and Object-Oriented Databases DOOD'89, Kyoto, Japan, 1989, pp. 535-554.

[Sch91] Schulte, D.: Ein Ansatz zur flexiblen Abbildung von Wissensmodellen auf Datenmodelle am Beispiel des Wissensmodells KOBRA und des Relationenmodells, Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1991.

[Sch93] Scholl, M.H.: Physical Database Design for an Object Oriented Database System, in: Vossen J.C., Maier D.E. (eds.), Query Processing for Advanced Databse Applications, Morgan Kaufmann, 1993.

[Sel79] Selinger, P. Griffiths, Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access Path Selection in a Relation– al Database Management System, Proc. ACM SIGMOD Conf., p. 23, Boston, MA, May–June 1979. Reprinted in M. Stonebraker, Readings in Database Systems, Morgan–Kaufman, San Mateo, CA, 1988

[SG88] Swami, A. and Gupta, A., Optimizing Large Join Queries, Proc. ACM SIGMOD Conf., p. 8, Chicago, IL, June 1988.

[SLR+92] Scholl, M.H., Laasch, C., Rich, C., Tresch, M., Schek, H.–J.: The COCOON Object Model, Technical Report 192, ETH Zürich, Dept. of Computer Science, 1992 .

[SLT91] Scholl, M., Laasch, C., Tresch, M.: Updatable Views in Object–Oriented Databases, Proc. Conf. on Deductive and Object–oriented Databases, Munich, Germany, December 1991.

[SPS87] Scholl, M., H.B. Paul, and H.J. Schek, Supporting Flat Relations by a Nested Relational Kernel, Proc. Int'l. Conf. on Very Large Data Bases, p. 137, Brighton, England, August 1987.

[SPSW90] Schek, H.J., Paul, H.B., Scholl, M.H., Weikum, G.: The DASDBS Project: Objectives, Experiences, and Future Prospects, IEEE Trans. on Knowledge and Data Eng., vol. 2, no 1, p. 25, March 1990.

[SS86] Schek, H.–J., Scholl, M. H.: The Relational Model with Relation–Valued Attributes, Information Systems, 11(2), p.137, June 1986.

[SS90a] Scholl, M. H., Schek, H.–J.: A Relational Object Model, Proc. ICDT, p. 89, Paris, France, December 1990, LNCS 470, Springer, Heidelberg.

[SS90b] Scholl, M. H., Schek, H.–J.: A Synthesis of Complex Objects and Object–Orientation, In Proc. IFIP TC2 Conf. on Object Oriented Databases (DS–4), Windermere, UK, July 1990. North–Holland, 1991

[St92] Strobel, M.: Konzeption einer Komponente zur Verwaltung von Kontexten im Wissensbankverwaltungssystem KRISYS, Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1992

[Sto92] Stoffel, M.: Erweiterung eines wissensbasierten Werkzeuges für den physischen Datenbank Entwurf, Diplomarbeit, ETH–Zürich, Departement Informatik, Zürich, 1992.

[Su91] Surjanto, B.: Entwurf und Implementierung eines wissensbasierten Systems zur Generierung eines anwendungsorientierten DB-Schemas für KRISYS Wissensbasen, Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1991

[Swa89] Swami, A., Research in Optimization of Large Join Queries, in Proc. Workshop on Database Query Optimization, ed. G. Graefe, p. 139, Oregon Graduate Center Computer Science Technical Report 89-005, Beaverton, OR, May 1989.

[TRSB92] Teeuw, W.B., C. Rich, M. H. Scholl, and H.M. Blanken, An Evaluation of Physical Disk I/Os for Complex Object Processing, Proc. IEEE Conf. on Data Eng. Vienna, Austria, April 1993. A more detailed version is available as Technical Report 183, ETH Zürich, Dept. of Computer Science, 1992.

[TS92] Tresch, M., Scholl M.H.: Meta Object Management and its Application to Database Evolution, in: Proc. the 11th Int. Conf. on Entity-Relationship Approach, Karlsruhe - Germany, 1992

[Wi84] Williams, C.: ART the Advanced Reasoning Tool: Conceptual Overview, Inference Corporation, Los Angeles, 1984.

[Wil91] Wilhelm, D.: Query–Schnittstelle für DASDBS, Diplomarbeit, ETH–Zürich, Departement Informatik, Zürich, 1991.

[WK90] Willshire, M. J., Hyoung-Joo, K: Proprties of Physical Storage Models for Object-Oriented Databases, in: Parbase 90: Proc. of Int. Conference on Databases, Parallel Architectures and their Application, Miami Beach, IEEE Computer Society Press, Los Alamos, 1990.

[WLH90] Wilkinson, K., Lyngbaek, P., Hasan, W.: The Iris Architecture and Implementation, IEEE Trans. on Knowledge and Data Engineering, 2(1), p.63, March 1990. Special Issue on Prototype Systems.