Proceedings

# 1986 International Workshop on Object-Oriented Database Systems

KLAUS DITTRICH AND
UMESHWAR DAYAL, EDITORS

Sponsored by: ACM Special Interest Group on Management of Data
IEEE CS Technical Committee on Database Engineering

In cooperation with: Gesellschaft fur Informatik, West Germany
FZI, University of Karlsruhe, West Germany
IIMAS, Mexico

IEEE Computer Society Order Number 734
Library of Congress Number 86-45866
IEEE Catalog Number 86TH0161-0
ISBN 0-8186-0734-3
ACM Order Number 472861

SEPTEMBER 23-26, 1986
ASILOMAR CONFERENCE CENTER
PACIFIC GROVE, CALIFORNIA

acm Association for Computing Machinery

THE COMPUTER SOCIETY OF THE IEEE

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC

IEEE COMPUTER SOCIETY PRESS

Proceedings

# 1986 International Workshop on Object-Oriented Database Systems

KLAUS DITTRICH AND
UMESHWAR DAYAL, EDITORS

Sponsored by: ACM Special Interest Group on Management of Data
IEEE CS Technical Committee on Database Engineering

In cooperation with: Gesellschaft fur Informatik, West Germany
FZI, University of Karlsruhe, West Germany
IIMAS, Mexico

IEEE Computer Society Order Number 734
Library of Congress Number 86-45866
IEEE Catalog Number 86TH0161-0
ISBN 0-8186-0734-3
ACM Order Number 472861

SEPTEMBER 23-26, 1986
ASILOMAR CONFERENCE CENTER
PACIFIC GROVE, CALIFORNIA



acm Association for Computing Machinery

THE COMPUTER SOCIETY OF THE IEEE

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE COMPUTER SOCIETY PRESS

Published by IEEE Computer Society Press
1730 Massachusetts Avenue, N.W.
Washington, D.C. 20036-1903

COVER DESIGNED BY JACK I. BALLESTERO

IEEE Computer Society Order Number 734
Library of Congress Number 86-45866
IEEE Catalog Number 86TH0161-0
ISBN 0-8186-0734-3 (paper)
ISBN 0-8186-4734-5 (microfiche)
ISBN 0-8186-8734-7 (case)
ACM Order Number 472861



Prices (1986) ACM or IEEE members: $22.50
All others: $45.00 prepaid

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
44 Hoes Lane
Piscataway, NJ 08854

ACM
Order Department
Post Office Box 64145
Baltimore, MD 21264

IEEE COMPUTER SOCIETY
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE
acm Association for Computing Machinery

# Workshop Committee

## PROGRAM COMMITTEE

Klaus Dittrich, FZI, University of Karlsruhe, Germany - Chairman
Umeshwar Dayal, Computer Corporation of America - Co-Chairman

Don Batory, University of Texas
Alex Buchmann, IIMAS, University of Mexico, Mexico
Mark Haynie, Instrumental Software
Dennis McLeod, University of Southern California

## CONFERENCE TREASURER

Dennis McLeod

## LOCAL ARRANGEMENTS

Mark Haynie

# Table of Contents

**Workshop Committee** ..... iii

**Opening Remarks**

Object-Oriented Database Systems: The Notion and the Issues ..... 2
*K.R. Dittrich*

**Session 1: Data Models and Constraints**

A Data Modeling Methodology for the Design and Implementation of Information Systems ..... 6
*P. Lyngbaek and W. Kent*

PDM: An Object-Oriented Data Model ..... 18
*F. Manola and U. Dayal*

CACTIS: A Database System for Specifying Functionally-Defined Data ..... 26
*S.E. Hudson and R. King*

A Generalized Constraint and Exception Handler for an Object-Oriented CAD-DBMS ..... 38
*A.P. Buchmann, R.S. Carrera, and M.A. Vazquez-Galindo*

**Session 2: Tools and Proposed Systems**

The Architecture of the EXODUS Extensible DBMS ..... 52
*M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita*

Object Management in POSTGRES Using Procedures ..... 66
*M. Stonebraker*

The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems ..... 73
*F. Maryanski, J. Bedell, S. Hoelscher, S. Hong, L. McDonald, J. Peckham, and D. Stock*

Extensibility in the Starburst Database System ..... 85
*P. Schwarz, W. Chang, J.C. Freytag, G. Lohman, J. McPherson, C.Mohan, and H. Pirahesh*

**Session 3: Languages and Interfaces**

A Strongly Typed, Interactive Object-Oriented Database Programming Language ..... 94
*A. Albano, G. Ghelli, M.E. Occhiuto, and R. Orsini*

Towards an Object-Centered Database Language ..... 104
*M.L. Kersten and F.H. Schippers*

Persistent and Shared Objects in Trellis/Owl ..... 113
*P. O'Brien, B. Bullis, and C. Schaffert*

How Helpful Is an Object-Oriented Language for an Object-Oriented Database Model? ..... 124
*O. De Troyer, J. Keustermans, and R. Meersman*

PROTEUS: Objectifying the DBMS User Interface ..... 133
*T.L. Anderson, E.F. Ecklund, Jr., and D. Maier*

## Session 4: Physical Aspects

Persistent Memory: A Storage Architecture for Object-Oriented Database Systems....148
*S.M. Thatte*
A Shared Object Hierarchy....160
*L.A. Rowe*
Indexing in an Object-Oriented DBMS....171
*D. Maier and J. Stein*
A Storage System for Complex Objects....183
*U. Deppisch, H.-B. Paul, and H.-J. Schek*
An Object Server for an Object-Oriented Database System....196
*A.H. Skarra, S.B. Zdonik, and S.P. Reiss*

## Position Statements

A Strongly Typed Persistent Object Store....206
*M.P. Atkinson, A. Dearle, and R. Morrison*
GENESIS: A Project to Develop an Extensible Database Management System....207
*D.S. Batory*
Sharing of Objects in an Object-Oriented Language....209
*A.-J. Berre*
Storage Reclamation for Object-Oriented Database Systems: A Summary of the Expected Costs....210
*M.H. Butler*
Combining Object-Oriented and Relational Models of Data....212
*R.G.G. Cattell and T.R. Rogers*
Identity and Versions for Complex Objects....214
*G.P. Copeland and S.N. Khoshafian*
The Use of Object-Oriented Databases to Model Engineering Systems....215
*C.M. Eastman*
New Approaches to Object Processing in Engineering Databases....217
*T. Härder*
An Object-Oriented Data Model for the Research Laboratory....218
*P. Kachhwaha and R. Hogan*
Inheritance Semantics for Computer-Aided Design Databases....219
*R.H. Katz*
Unifying Database and Programming Language Concepts Using the Object Model....221
*A.M. Keller*
Object-Oriented Data Models and Management of CAD Databases....223
*M.A. Ketabchi*
Communicating Recursive Objects....225
*W. Lamersdorf*
Why Object-Oriented Databases Can Succeed Where Others Have Failed....227
*D. Maier*
Object Management and Sharing in Autonomous, Distributed Databases....228
*D. McLeod*
Transaction Management for Object-Oriented Systems....229
*J.E.B. Moss*
Re-creation and Evolution in a Programming Environment....230
*J.R. Nestor*

Object Modeling ..... 231
*S. Osborn*
An Object-Oriented Framework for Modeling Design Data ..... 232
*K.V. Bapa Rao*
An Object-Oriented Data Management System for Mechanical CAD ..... 233
*D.L. Spooner*
The GRASPIN DB—A Syntax Directed, Language Independent Software Engineering Database ..... 235
*C. Zaroliagis, P. Soupos, S. Goutas, and D. Christodoulakis*

**Author Index** ..... 237

# Opening Remarks

# Object-oriented Database Systems: the Notion and the Issues

(extended abstract)

Klaus R. Dittrich

Forschungszentrum Informatik (FZI) an der Universität Karlsruhe
Haid - und - Neu - Str. 10-14, D-7500 Karlsruhe

A *database system* is a collection of stored data together with their description (the *database*) and a hardware/software system for their reliable and secure management, modification and retrieval (the *database management system*, DBMS).

A database is supposed to represent the interesting semantics of an application (the *miniworld*) as completely and accurately as possible. The *data model* incorporated into a database system defines a framework of concepts that can be used to express the miniworld semantics.
It comprises

- basic data types and constructors for composed data types,
- (generic) operators to insert, manipulate, retrieve and delete instances of the actual data types of a database,
- implicit consistency constraints as well as (eventually) mechanisms for the definition of explicit consistency constraints that further reflect the miniworld semantics as viewed by the database system.

As usual, types have to be defined before instances of them can be created (the collection of defined types – sometimes together with the set of explicit consistency constraints – forms the *database schema* ). Every database thus adheres to the schema defined for it, and both together, the schema and the actual data provided by the users (and stored in instances) capture the miniworld semantics.

We can therefore distinguish the following two classes of semantics:

- the semantics of the miniworld itself,
- the semantics of the miniworld as represented within the database.

Let us assume that a database *correctly* reflects the intended miniworld semantics (careful database design! ). Due to the rigid framework of data models, there will still remain a semantic gap between the miniworld and its database representation. In other words, it is usually impossible to represent *all* interesting semantics within a database. The "remainder" has to be captured by the application programs using the database and/or it is part of the (hopefully meaningful ! ) interpretation of the result of database queries by the user himself.

However, the ultimate goal of database systems is to provide for concepts that allow to keep the semantic gap as small as possible and thus permit to represent most of the salient semantics in the database itself.

**What are object-oriented database systems ?**

This is where object-oriented database systems come in. On a level of abstractions, the semantics of a given miniworld may be modelled as a set of entities and relationships amongst them. While classical business/administration types of database applications tend to deal with rather "simple" entities (i.e. those where only a few properties like name, age, salary are of interest), this is no longer true for applications like VLSI-design, image processing, office automation and the like. In these areas, entities usually show very complex internal structures and may comprise larger numbers of (possibly substructured) properties. Similar observation can be made with respect to relationships.

Today's database systems are mostly based upon one of the now classical data models (hierachical, network, or relational) or one of their derivations. As these models are all tailored to account for the representation of rather simple entities only, the semantic gap tends to become large when complex entities need to be dealt with. This is at least partially due to the fact that in these cases *one* conceptual miniworld entity has to be represented by *a number* of database objects (e.g. records, tuples, ...).

This leads us to a first-cut informal definition of an object-oriented database system: it is based on a data model *that allows to represent one miniworld entity* (whatever its complexity and structure) *by exactly one object* (in terms of the data model concepts) *of the database*. Thus no artificial decomposition into simpler concepts is necessary in any case (unless the database designer decides to do so). Note that as entities might be composed of subentities which are entities in their own right, an object-oriented data model also has to allow for recursively composed objects.

Looking at things a little closer, we can in fact identify several *levels of object-orientation*:

(a) if the data model allows to define data structures to represent entities of any complexity, we call it *structurally object-oriented* (i.e. there are complex objects),

(b) if the data model includes (generic) operators to deal with complex objects in their entirety (in contrast to being forced to decompose the necessary operations into a series of simple object – e.g. tuple or homogeneous set of tuples – operations), we call it *operationally object-oriented*; as it is hardly meaningful without, we require that operational object-orientation includes structural object-orientation;

(c) borrowing types from the object-oriented programming paradigm, a data model may also incorporate features to define object types (again of any complexity) together with a set of specific operators (abstract data types); instances can then only be used by calling these operators, their internal structure may only be exploited by the operator implementations; we call systems based on this approach *behavioralley object-oriented.*

While structural object-orientation is not very useful without operational object-orientation, both (b) *and* (c) are within the range of object-oriented database systems; note that the scope is thus broader than with object-oriented programming languages which are concentrated on the paradigm sketched in (c) only. However, there seems to be agreement that even the (somewhat less advanced) operational object-orientation is a versatile solution for database systems (and might at least be used as a basis for the internals of abstract data types in (c) ). By the way, concepts like property inheritance may be added to both (b) and (c).

There are at least two other directions where the notion of object-orientation is used; in our opinion, they do not contribute to the definition of object-oriented database systems, but naturally come along with such systems.

- *object-oriented implementation* : the (database) system as a piece of software is constructed as a set of abstract data type instances, i.e. a specific kind of modularization is applied, even non-object-oriented database systems may have an object-oriented implementation;
- *object-oriented user/programming interface* : the database system interface is presented to the user/application programmer in a fashion inspired by the object - oriented proramming paradigm; while this fits in very smoothly with an object - oriented database system (expecially of type (c) ), it may also be provided on top of any other database system.

## What are the issues of object-oriented database systems ?

At first glance, one might think that object-oriented database systems just offer a different kind of data model than traditional systems do. However, the more powerful concepts for modelling miniworld semantics result in a number of database issue to be at least reconsidered, if not extended or completely changed. The following list mentions a few of them in random order:

- Like various record-/tuple-oriented data models, many object-oriented data models have been and will be proposed. Some of these differ only slightly in style and/or expressive power, and there is currently no clear tendency towards one or a small number of generally "recognized" models.
- Together with complex objects, applications are often concerned with *object versions* (multiple representations of the same semantic entity, to account for different stages, different times of validity, alternative or hypothetical information etc.). Object-oriented database systems therefore need mechanisms to deal with versions.
- When manipulating objects comprising large bulks of data, transactions may become much longer than usual. New concepts are therefore needed to accommodate long-duration transactions, and in addition the concepts for recovery and consistency control and their relationship to the transaction concept have to be reconsidered.
- Protection mechanisms have to be based on the notion of object which is the natural unit of access control in this framework.
- For databases containing large numbers of data, archiving may become a major issue. Again, objects (and their versions, if any) form the natural unit for this activity.
- To work with an object-oriented database offen consists of first selecting one or a small number of objects and then performing local operations on them for a while. This suggests to provide, among others,
  - specialized access paths for complex objcts,
  - specialized storage structures for complex objects (that e.g. physically cluster logical objects or that use delta storage for versions),
  - object-oriented main memory buffering

  in the implementation of an object-oriented database system.
- High quality database design is a tedious job even for record-oriented database systems. It appears that it is even more difficult within the framework of object-oriented database systems. Appropriate design methodologies and tools that support them have to be developed.

# Session 1: Data Models and Constraints

# A DATA MODELING METHODOLOGY FOR THE DESIGN AND IMPLEMENTATION OF INFORMATION SYSTEMS

*Peter Lyngbaek and William Kent*

*Hewlett-Packard Laboratories*
*1501 Page Mill Road, Palo Alto, California 94304*

ABSTRACT

Formal specifications that precisely and correctly define the semantics of software systems become increasingly important as the complexity of such systems increase. The emerging set of semantic data models which support both structural and operational abstractions are excellent tools for formal specifications. In this paper we introduce a methodology, based on an object-oriented data model, for the design and development of large software systems. The methodology is demonstrated by applying the object-oriented data model to the specification of a database system which implements the given model. The specification serves several purposes: it formally defines the precise semantics of the operations of the data model, it provides a basis from which the corresponding database system software can be systematically derived, and it tests and demonstrates the adequacy of such a model for defining software systems in general. The design methodology introduced combines techniques from data modeling, formal specifications, and software engineering.

## 1 Introduction

The importance of formal semantics definitions of large, complex software systems has been widely recognized over the past years. A formal semantics definition is a precise specification of the semantics of a given system. It serves as a basis for the development of implementations of the system and can be used to verify correctness of the implementations. A formal definition can be thought of as a contract between users and developers of the system being defined.

Operational and denotational semantics[4,16] have been demonstrated to be useful tools for programming language design and compiler developments[3,5,14] and there is a growing acceptance of such techniques for data model and database system designs[6]. A formal definition of a database system expresses the semantics of the database operations and it may be used by the implementors of the database system, technical manual writers, database designers and database end-users.

Data models are becoming increasingly powerful. Their semantic expressiveness and high levels of data abstractions make the differences between semantic data models[1,9,24,17,25] and modern object-oriented programming languages[15] vanish. Data models that support both static (structural) and dynamic (operational) modeling constructs can be used as formal specification tools suitable for semantics definitions of complex software systems such as information systems.

In this paper, we show how to use an object-oriented data model as a tool for specifying large software systems. The approach, which supports a data-driven methodology for the design and development of software systems, is an extension of the fact-based data analysis and design described in [19]. The methodology is demonstrated by defining an object-oriented database system in terms of the data model it implements. The static modeling constructs of the data model, e.g., objects, types, and relationships, can be used to define the semantic domains of the corresponding information system. The dynamic modeling constructs, e.g., the database operations, can be used to define the semantics of the operations of the information system. The definition is based on a procedural programming language similar to the ones a database designer might use for writing database operations.

The advantages of using a data model as a tool for the definition of its own semantics are twofold. Firstly, a user of the formal definition only has to deal with one formalism, namely the data model itself. An application developer learns the capabilities of the database system in the same formalism he will use to define his own applications. An understanding of the data model is required anyway by the system implementors, manual writers, database designers, and database users in order for them to do their jobs. The fact that the user of the formal definition does not have to understand an additional specification language is an important one because the major objections to formal specification languages are their complexity and difficulty of use and understanding. The second reason for using a data model as a formal specification tool is to demonstrate the power of the model. This is similar to implementing a compiler for a given programming language in its own language.

To a limited extent, data models have previously been used to specify themselves. Meta-data that describe the structure of a database can be modeled by the same model as the one used to model the content of the database. Various implementations of the relational model, for example, store the meta-data in pre-defined relations[8,27]. These relations, often called the system tables, describe all the relations of a database, including themselves, by their names, attributes, access rights, and so on. A number of semantic data models have also been used to describe their own meta-data[10,7,22]. In the object-oriented data model supported by the Personal Data Manager[22], for example, a type is modeled as an object with pre-defined attributes such as Name, Instances, Supertype, and Subtypes. Self-describing database systems[23] introduce an intension-extension dimension of data description which allows changes of user-data explicitly to be controlled in the corresponding schema. That way, the same data manipulation language can be used to manipulate both meta-data and user-data.

Even though it has been successfully demonstrated that the structure of a database can be defined in terms of its own structural modeling constructs, little work has been done to demonstrate how the semantics of the data definition and manipulation operations of a database system can be defined in terms of the primitive operations of the database system. In [6], it is shown how the structure of an IMS database as well as the semantics of the IMS database commands can be modeled using the VDL notation[4]. The goal of this paper is to illustrate the same thing for an object-oriented database system, but instead of using VDL as a formal specification language we will use the structural and operational modeling constructs of the underlying object-oriented data model.

The rest of this paper is organized as follows. Section 2 describes the object-oriented data model, called the Iris Data Model, which provides the framework for the formal specification methodology. Section 3 outlines the design methodology and illustrates how it can be applied to the specification of the Iris database management system[13]. Finally, Section 4 contains some concluding remarks.

## 2 Iris Data Model Overview

The notion of object or entity is central to most semantic data models[1,9,24,17,25]. Object-oriented data models introduce a semantically rich set of structuring primitives that support abstractions such as classification, generalization/specialization, and aggregation[26]. Objects, which represent things or concepts from an application environment, are unique entities in the database with their own identity and existence. They can be referred to regardless of their attribute values. Therefore, referential integrity[11] can be supported. This is a major advantage over record-oriented data models in which the objects, represented as records, can be referred to only in terms of their attribute values.

Semantic models that support the modeling of database operations, i.e., procedural abstractions[24,2,21,12] introduce a high degree of data independence. In such models, objects are described - not by their looks - but by their behavior. Objects can only be accessed and manipulated in terms of pre-defined operations and functions.

The Iris Data Model falls into the general category of semantic data models. The roots of the model can be found in previous work on Daplex[25] and its extensions[20] and the Taxis language[24]. A subset of the Iris model, which is currently being implemented at Hewlett-Packard Laboratories, is briefly described below. It is beyond the scope of this paper to further compare the model with related work on semantic data modeling.

### 2.1 Objects, Types, and Functions

The data model is based on objects, types, and functions. Objects have the following characteristics:

1. Objects are classified by types. Objects that share common properties belong to the same type.

2. Objects may serve as arguments to functions and may be returned as results of functions.

The model distinguishes between literal objects and non-literal objects. Literal objects include Integer, Real, Boolean, and String objects. They are directly representable. Literal objects are system-defined and are always known to the database. I.e., there are no operations to explicitly create or destroy them. Non-literal objects are not directly representable in external form. Internally, non-literal objects are represented by surrogates which are unique object identifiers.

The database operation *NewObject* introduces a new object and adds it to the extension of a specified user-defined type and all its supertypes. The database operation *DeleteObject* deletes a specified user-defined object from the database.

Types, which have unique names, are organized in a type structure that supports generalization and specialization. A given type may have multiple subtypes and multiple supertypes. An object that belongs to a given type also belongs to the type's supertypes. The type Object is the supertype of all other types and therefore contains every object. Types are objects themselves, and their relationships to subtypes, supertypes, and instances are expressed as functions in the system[18].

For each type in the database there is an associated predicate function, called a *typing function*, which has the same name as the type. The typing function maps objects onto the Boolean objects True and False. A given object is mapped to True if it is an instance of the type with which the typing function is associated; otherwise it is mapped to False.

The database operation *NewType* introduces a new type of a specified name as a subtype of specified supertypes. An existing user-defined type is deleted from the database by the database operation *DeleteType*. The operation *AddInstance* adds a specified user-defined object to the extension of a specified user-defined type and all its supertypes. A user-defined object is removed from the extension of a specified user-defined type and all its subtypes by the database operation *RemoveInstance*.

Properties of objects are expressed in terms of (possibly multi-valued) functions, which are defined over types. For example, DepartmentOf is a function defined on Employee objects:

```
DepartmentOf:  Employee -> Department
```

DepartmentOf(Smith) will return the department to which Smith is assigned, e.g., Sales.

A type can be characterized by the collection of functions defined on it. The Employee type might have the functions EmployeeNumber, Name, DepartmentOf, Salary, and Birthdate defined over it:

```
EmployeeNumber:  Employee -> Integer
Name:  Employee -> String
DepartmentOf:  Employee -> Department
Salary:  Employee -> Real
Birthdate:  Employee -> Date
```

Functions can also express properties of several objects. For example, the function AssignmentDate defined on employees and departments will return the date an employee of a given department was assigned to that department:

```
AssignmentDate:  Employee x Department ->
   Date
```

If Smith was assigned to the Sales department on 1/1/84 then

```
AssignmentDate(Smith, Sales) = 1/1/84
```

Functions may have complex property values. The function AssignedOn defined on Date objects returns pairs of Employee and Department objects:

```
AssignedOn:  Date -> Employee x Department
```

If two assignments were made on 6/1/82, e.g., Wong was assigned to the Research department and Jones to the Marketing department then

```
AssignedOn(6/1/82) =
   [(Wong, Research), (Jones, Marketing)]
```

The AssignedOn function illustrates a multi-valued function.

A function is defined not only on the types explicitly mentioned in the function definition, but also on the subtypes of those types. This is referred to as inheritance. For example, if the type Engineer is a subtype of Employee then the functions EmployeeNumber, Name, DepartmentOf, Salary, and Birthdate are automatically defined on the Engineer type.

The database operation *NewFunction* introduces a new function and the operation *DeleteFunction* deletes a specified function from the database.

### 2.2 Database Updates and Queries

Properties of objects can be modified by changing the values of functions, e.g.,

```
Set Salary(Smith) = $30000.00
```

The values of multi-valued functions can also be modified by Add and Remove operations.

The database can be queried by specifying predicates on objects and function values. The following operation retrieves all the employees assigned to Sales on 3/1/83:

```
FIND e/Employee
WHERE AssignmentDate(e, Sales) = 3/1/83
```

and the operation

```
FIND d/Department
FOR SOME e/Employee
WHERE AssignmentDate(e, d) = 3/1/83
```

returns all the departments to which employees were assigned on 3/1/83.

A FIND operation without a list of return variables returns a Boolean value. The value returned is True if there exists a binding of the existentially quantified variables such that the Boolean expression is True. For example, the FIND operation

```
FIND FOR SOME d/Department
WHERE DepartmentOf(Jones) = d
```

returns the value True if Jones is assigned to a department. If Jones is not employeed by any department the value False is returned.

### 2.3 Function Interdependence

Some functions are semantically interdependent, in the sense that an update to one should be reflected by a change in the other. Inverse functions constitute the most common example. Consider the properties of a department:

```
Name:  Department -> String
Manager:  Department -> Employee
Employees:  Department -> Employee
```

At present, we expect the value of Employees(Sales) to include Smith. But suppose we perform the update

```
Set DepartmentOf(Smith) = Purchasing
```

Such an update should automatically update the Employees function so that Smith will appear among the employees of Purchasing rather than Sales.

Sometimes more than two functions are so inter-related. All of the following functions are semantically interrelated:

```
AssignmentDate:  Employee x Department ->
   Date
AssignedOn:  Date -> Employee x Department
DeptHist:  Department -> Employee x Date
EmpHist:  Employee -> Department x Date
```

The function DeptHist returns for a given department its past and current employees and their assignment dates. Similarly, EmpHist returns for a given employee his employment history.

The interdependence among such functions is expressed by deriving them from a common underlying base predicate. Predicates are Boolean-valued functions that express relationships among the objects involved. For example, an Assignment predicate expresses a relationship among employees, departments, and dates.

```
Assignment:
   Employee x Department x Date -> Boolean
```

If Smith was assigned to the Sales department on 1/1/84 then

```
Assignment(Smith, Sales, 1/1/84) = True
```

The functions AssignmentDate, AssignedOn, EmpHist, and DeptHist can then be defined by the following derivations:

```
AssignmentDate(e, d) ::=
   FIND date/Date
   WHERE Assignment(e, d, date)

AssignedOn(date) ::=
   FIND e/Employee, d/Department
   WHERE Assignment(e, d, date)

DeptHist(d) ::=
   FIND e/Employee, date/Date
   WHERE Assignment(e, d, date)

EmpHist(e) ::=
   FIND d/Department, date/Date
   WHERE Assignment(e, d, date)
```

In effect, an update to one of these functions implies an update to the underlying predicate, which in turn propagates into all the other functions derived from that predicate. Notice that a function derivation provides a definition for a function previously introduced by a NewFunction operation.

The functions DepartmentOf and Employees are also semantically related to Assignment. It is fairly easy to see how the functions DepartmentOf and Employees can be derived from the predicate

```
CurrentAssignment:
   Employee x Department -> Boolean
```

which relates employees to their current departments. CurrentAssignment can be derived from the base predicate Assignment by the following derivation:

```
CurrentAssignment(e, d) ::=
   FIND WHERE FOR SOME date1/Date
   (Assignment(e, d, date1) AND NOT
   FOR SOME date2/Date, d1/Department
   (Assignment(e, d1, date2) AND
   date2 > date1))
```

### 2.4 Object Participation Constraints

Functions of one argument are generally characterized as being either single-valued or multi-valued, and either required or optional. "Participations" express such constraints in a more generalized fashion, applicable to functions of multiple arguments, including predicates. In the future, participations will also be specifiable for sets of argument positions, expressing constraints on combinations of argument values.

Functions have specifications for each of their argument and result parameters which indicates a minimum object participation (MINP) and a maximum object participation (MAXP). The object participation specifications, which are required by every Iris function, are constraints. As an example, consider the CurrentAssignment predicate in which both MINP and MAXP of Employee objects are one, and MINP and MAXP of Department objects are zero and many, respectively:

```
CurrentAssignment:
  Employee [1,1] x Department [0,m] -> Boolean
           a b                 c d
```

(a) Each employee must participate at least once (MINP = 1), i.e., each employee must have a department.

(b) Each employee may participate at most once (MAXP = 1), i.e., each employee must have a department.

It is useful to note that any parameter having MAXP = 1 can serve as a unique key in the underlying tables in which the data is stored. This applies to employees in this case, since no employee can occur here more than once.

(c) A department need not participate at all (MINP = 0), i.e., a department may have no employees.

(d) A department may participate many times (MAXP = m), i.e., a department may have many employees.

The minimum and maximum object participation constraints for a base predicate implicitly determine the object participation constraints for all the derived functions whose derivations depend on the predicate.

For ordinary functions of one argument, participations correspond to simpler constraints. Consider the DepartmentOf function:

```
DepartmentOf:
   Employee [1,1] -> Department [0,m]
            a b
```

(a) Each employee must participate at least once (MINP = 1), hence this function is "required". An argument with MINP = 0 would indicate an optional function.

(b) Each employee may participate at most once (MAXP = 1), hence this function is single-valued. An argument with MAXP = m would indicate a multi-valued function.

### 2.5 Database Design Operations

A particular implementation of the Iris data model may support a number of database design operations which allow physical storage structures and access methods for functions to be defined. The Iris prototype currently being developed supports two database design operations, *Store* and *Index*, described below.

#### *2.5.1 Store*

The Store operation specifies how to materialize one or more predicate functions. Store implements base predicate functions using tables, i.e., the graphs of the functions are stored in tables. That way, the database operations Set, Add, Remove, and Find are mapped by the database system to table operations.

In order to illustrate the Store operation, consider the three predicates

```
Employee:  Object [1,1] -> Boolean
Project:
   Employee [0,1] x Project [0,m] -> Boolean
Manager:
   Subordinate/Employee [0,1] x
   Supervisor/Employee [0,m] -> Boolean
```

The operation

```
Store Project
```

causes a single table to be created and used to materialize the Project predicate function. The table has one column for each of the argument parameters Employee and Project. The table will only contain employees who are assigned to projects and projects to which employees are assigned.

If several predicates are specified by the same Store operation, they will all be clustered together in a single table. The operation

```
Store Project ON Employee, Manager ON
   Subordinate
```

causes the two predicates Project and Manager to be implemented using the same table. The table has a single column, called the clustering column, which is shared by the parameters Employee of Project and Subordinate of Manager. These parameters, called the clustering parameters, are specified by the ON clauses of the Store operation. All the clustering parameters must have the same object type and they must be "keys", i.e., their upper object participation must be one. In addition to the clustering column, the table has one column corresponding to each of the other parameters of the specified predicates. The clustering column will only contain employees who are assigned to projects or have managers. If an employee is assigned to a project, but has no manager, the manager value will be null. Correspondingly, if an employee has a manager, but is not assigned to a project, the project value will be null.

It is possible to cluster the typing function of a given type with other predicates if the type coincides with the clustering parameters. Thus, the operation

```
Store Employee, Project ON Employee, Manager
   ON Subordinate
```

causes the instances of the Employee type, the Project predicate, and the Manager predicate to be stored together in the same table. The table has three columns, one corresponding to Employee objects, one to Project objects, and one to Supervisor (also Employee) objects. Now the clustering column will contain all employees, with unassigned employees having a null project and a null supervisor.

When a function stored with an ON clause is a unary predicate, a column is generated to contain the Boolean value of the predicate. Consider, for example, the predicate

```
Exempt:  Employee [1,1] -> Boolean
```

The following Store operation will create a table with two columns:

```
Store Employee, Exempt ON Employee
```

The clustering column is of type Employee. The other column, which corresponds to the predicate Exempt, is of type Boolean.

#### *2.5.2 Index*

The Index operation creates search indexes for the tables created by the Store operation. The operation

```
Index Project ON Employee
```

causes an index to be created for the table used to implement the Project predicate. The index is defined on the column which corresponds to the Employee parameter of Project.

### 2.6 Modularization

The full Iris Data Model will eventually support a modularization mechanism that allows related functions to be grouped together in modules. Functions visible within a module can then be specified to be invisible outside the module or visible only in certain other modules. That way,