Mapping Object-Protocol Schemas into Relational Database Schemas and Queries *

OPM Version 4.1

I-Min A. Chen[†]

Victor M. Markowitz[‡]

Data Management Research and Development Group
Information and Computing Sciences Division
Lawrence Berkeley National Laboratory
1 Cyclotron Road, Berkeley, CA 94720

June 1996

*Issued as Technical Report LBNL–33048. This work is supported by the Office of Health and Environmental Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

[†]Author's e-mail address: *IAChen@lbl.gov*, phone: (510) 486-7264, fax: (510) 486-4004
[‡]Author's e-mail address: *VMMarkowitz@lbl.gov*, phone: (510) 486-6835, fax: (510) 486-4004

Abstract

The Object-Protocol Model (OPM) is an object data model that supports modeling scientific (e.g., genomic) database applications in terms of objects and protocols (laboratory experiments). Objects and protocols are qualified by attributes that are defined over (attribute) value classes.

OPM is currently implemented on top of relational database management systems (DBMSs). For this purpose, OPM schemas and queries are mapped into abstract (DBMS-independent) relational schemas and queries, and then into DBMS-specific relational database schema definitions and SQL queries. The mapping of OPM schemas into abstract relational schemas and queries is described in this document.

This document describes the mapping of OPM version 4.x schemas into relational schemas, queries, and procedures. OPM version 4.x supports the following additional features (not supported in versions 1.x, 2.x and 3.x):

- Attribute null constraints (`can be null` and `not null`) are replaced by attribute cardinality (`max`, `min`) constraints.
- Attributes can be associated with `order by` clauses in the schema definition.
- Primitive attributes can be associated with default values.
- `REP` attributes can be versioned.
- The length limitation (255 characters) has been removed for descriptions and properties; descriptions and properties can have unlimited length.
- Derived subclasses and derived superclasses are supported.
- A new type of *user specified* derived attribute has been added.
- System class `OBJECTS_InClass` no longer exists.
- Physical database features can be specified using class and attribute properties through special keywords.

OPM version 4.1 supports the following new features (not supported in version 4.0):

- abbreviated composition path expressions;
- derived attribute definitions can contain `count` applied on other composition derived attributes;
- `ORDER BY` for set-valued attributes;
- `EXAMPLE` specifications for classes and attributes;
- support for modeling permissions in terms of object ownership and status (e.g., released, not released).

Contents

1 Introduction **1**

2 Preliminary Definitions and Mapping Steps **3**
2.1 Relational Constructs 3
2.2 Preliminary Mapping Steps 5

3 Mapping OPM Structures and Constraints **9**
3.1 Mapping Outline and Motivation 9
3.2 Mapping Details 12
3.2.1 Mapping Metaclasses 13
3.2.2 Mapping Named Primitive and Controlled Value Classes 14
3.2.3 Mapping Classes 14
3.2.4 Mapping Attributes 15
3.3 Mapping Derived Classes 18

4 Retrieval and Data Manipulation Methods **23**
4.1 Representing Methods 23
4.2 Retrieval Methods for Non-Derived Attributes 23
4.3 Retrieval Methods for Arithmetic or Aggregate Function Derived Attributes 29
4.4 Retrieval Method for Composition Derived Attribute 30

5 Data Manipulation Methods **42**
5.1 Insert Method 42
5.2 Deletion Methods 44
5.3 Logical Delete Method 45
5.4 Logical Undelete Method 45
5.5 Attribute Value Update Methods 46
5.6 Set Default Version Method 46

6 The Effect of Permission Modeling on Methods **47**
6.1 Get Owner 47
6.2 Update Owner 47
6.3 Get Status 48
6.4 Update Status 48

7 Conclusion **49**

References **50**

A The Syntactic Definition for OPM Schemas **51**
A.1 BNF Grammar for OPM 51
A.2 Additional Constraints for OPM Schemas 65

B The Syntactic Definition for Relational Schemas **71**

1 Introduction

In [Chen & Markowitz 96a] we have defined a data model called the *Object-Protocol Model* (OPM), that allows modeling scientific (e.g., genomic) database applications in terms of objects and protocols (laboratory experiments). OPM is currently implemented on top of relational database management systems (DBMSs). In order to implement OPM on top of a relational DBMS, schemas specified in OPM have to be mapped into DBMS-specific relational schema definitions and SQL queries. Our approach to this mapping is to use an intermediate abstract (DBMS-independent) relational level, so that OPM schemas are mapped first into abstract relational schemas and queries, and then into relational DBMS-specific database schema definitions and queries. In this document we describe the mapping of OPM schemas into abstract relational schemas and queries. This mapping is implemented as part of an OPM schema translator [Chen & Markowitz 96b].

OPM is a semantic data model [Hull & King 87], closely related to object-oriented data models [ODMG-93], that supports the specification of object classes and protocol classes. An object class is identified by a class name, has an optional class description, and is associated with attributes that qualify the objects (instances) of the class. A subset of the attributes can be specified as the identifier for distinguishing among the objects of a class.

Attributes in OPM can have values that are atomic or consisting of a tuple of atomic values, and can be single-valued, set-valued, or list-valued. An attribute takes values from value classes that are either other classes or system provided primitive value classes such as `INTEGER`. Abstract attributes can take values from (local) classes specified in the current OPM schema or specified in other (foreign) schemas; this facility allows referencing data across multiple databases.

Attributes can be generic or versioned. Generic attributes can be used for describing the stable (non evolving) part of an object or protocol (such as the social security number of a person), while versioned attributes can be used for describing the changing (evolving) part of an object (such as the address of a person). A version of an object or protocol x consists of the values of the generic attributes for x (common to all versions of x) and the values of versioned attributes for a specific version of x.

OPM supports the specification of subclass-superclass relationships in an object class (ISA/ISA*) hierarchy. A subclass is a specialization of its superclasses, and inherits all the attributes associated with its superclasses. OPM supports multiple inheritance. OPM also supports the specification of derived attributes using derivation rules involving arithmetic expressions, aggregate functions, and attribute composition.

Protocol classes in OPM are used to model laboratory experiments. Given an input, a protocol instance (experiment) results in an output, where both input and output consist of objects. OPM supports the recursive specification (expansion) of protocols. Protocol expansion in OPM allows specifying a protocol in terms of alternative subprotocols, sequences of subprotocols, and optional protocols. A protocol class can be associated with regular as well as `input` and `output` attributes. Input and output attributes are used for specifying input and output connections between protocols.

OPM supports derived object classes. There are two types of derived classes: specialization derived classes and generalization derived classes. A specialization derived class (or derived subclass) is defined as a subclass of another derived or non-derived object class with an optional derivation condition. A generalization derived class (or derived superclass) is defined as a union of two or more derived or non-derived object classes.

OPM 4.1 also supports modeling permissions in terms of object ownership and status. `OWNER` and `STATUS` attributes can be defined for all the classes with 'permissions on' in the schema. The owner of an object can retrieve and update this object; non-owners can only retrieve released objects.

The rest of this document is organized as follows. Preliminary definitions are given in section 2. The detailed mapping of OPM schemas into relational schemas and queries is defined in sections 3, 4, and 5. The support of permission model in queries and updates is described in section 6. Section 7 contains concluding remarks. The syntactic definition for OPM schemas is given in Appendix A. Appendix B contains the syntactic definition for relational schemas.

2 Preliminary Definitions and Mapping Steps

In this section, we briefly review the relational concepts used for specifying the mapping of OPM schemas, and present the preliminary steps of this mapping. The relational data model is described in detail in [Maier 83]. OPM constructs are described in detail in [Chen & Markowitz 96a] and therefore are not reviewed in this document. An example of an OPM schema is shown in figure 1.

2.1 Relational Constructs

The relational data model provides a very simple construct for representing data structures, a tabular structure called relation consisting of columns called attributes, where each attribute is atomic (non-decomposable) and single-valued. The relational data model also provides a large range of constraints that express inner relation or inter relation data dependencies (e.g., see [Maier 83]). We consider only constraints that are supported by commercial relational DBMSs and that are necessary for representing the constructs and constraints of OPM.

Null constraints are used for restricting attribute values to non-null values. Range constraints are used to restrict the allowed range of values for attributes. Referential integrity constraints have been added to most commercial DBMSs only recently and are used for expressing existence dependencies between tuples: such a constraint is specified by associating a *foreign* key in a (referencing) relation with the *primary* keys of a list of (referenced) relations, and requires the foreign-key values in tuples of the referencing relation to match primary-key values in tuples of one of the referenced relation [Date 86]; if the list of referenced relations consists of a single relation, then the referential integrity constraint is called *single-target*, otherwise it is called *multi-target*. Finally, exclusion constraints are used for expressing mutual exclusion of keys in different tuples: similar to a referential integrity constraint, such a constraint is specified by associating a *foreign* key in a (referencing) relation with the *primary* keys of a list of (excluded) relations, and requires the foreign-key values in tuples of the referencing relation not matching any primary-key value in tuples of any of the excluded relation.

A referential integrity constraint is associated with a restricted insert rule and a restricted, cascades, or nullifies delete rule [Date 86]. For a referential integrity constraint involving referencing relation R_i and referenced relation R_j:

(i) the restricted insert rule asserts that inserting a tuple t into R_i can be performed only if the tuple of R_j referenced by t already exists;

(ii) a restricted delete rule asserts that the deletion of a tuple t' from R_j cannot be performed if there exist tuples in R_i referencing t';

(iii) a cascades delete rule asserts that the deletion of a tuple t' from R_j implies deleting the tuples of R_i referencing t'; and

```
SCHEMA Example
   DESCRIPTION: "an example schema"
   PERMISSIONS
      OWNER owner: PERSON
      STATUS status: INTEGER
                     DEFAULT: 0
CONTROLLED VALUE CLASS RANK {"Professor","Associate Professor","Assistant Professor"}
CONTROLLED VALUE CLASS AGE      { 0 - 120 }
OBJECT CLASS PERSON
   PERMISSION OFF
   ID: SSN
   REP: Name
   ATTRIBUTE SSN: [1,1] INTEGER
   ATTRIBUTE Name: [0,1] VARCHAR(50)
   ATTRIBUTE Age: [0,1] AGE
OBJECT CLASS FACULTY    isa PERSON
   ATTRIBUTE Rank: [0,1] RANK
   ATTRIBUTE Department: [0,1] DEPARTMENT      inverse of DEPARTMENT.Faculty
   VERSIONED
      ATTRIBUTE Teaches (Course, InDepartment):
                         set-of [0,] ([0,1] COURSE, [0,1] DEPARTMENT)
                         Course inverse of COURSE.Taughtby
      ATTRIBUTE CourseBooks   DERIVATION: !Taughtby [COURSE] References [BOOK] Title
DERIVED OBJECT CLASS PROFESSOR
   PERMISSION OFF
   DERIVATION: subclass of FACULTY
   CONDITION: [FACULTY] Rank = "Professor";
OBJECT CLASS COURSE
   PERMISSION OFF
   ID: Name
   REP: Name
   ATTRIBUTE Name: [1,1] CHAR(10)
   VERSIONED
      ATTRIBUTE Taughtby: [0,1] FACULTY
      ATTRIBUTE References: set-of [0,] BOOK or ARTICLE
      ATTRIBUTE BookAuthors   DERIVATION: References [BOOK] Authors [VARCHAR(80)]
OBJECT CLASS DEPARTMENT
   PERMISSION OFF
   ID: Name
   REP: Name
   ATTRIBUTE Name: [1,1] VARCHAR(50)
   ATTRIBUTE Faculty: set-of [0,] FACULTY      inverse of FACULTY.Department
                      DELETE NULLIFIES
OBJECT CLASS BOOK
   ID: ISBN
   REP: ISBN, Title
   ATTRIBUTE ISBN: [1,1] CHAR(30)
   ATTRIBUTE Title: [0,1] VARCHAR(50)
   ATTRIBUTE Authors: list-of [0,] VARCHAR(80)
OBJECT CLASS ARTICLE
   ID: Title
   REP: Title
   ATTRIBUTE Title: [1,1] VARCHAR(30)
   ATTRIBUTE Authors: list-of [0,] VARCHAR(80)
```

Figure 1: An OPM Schema Example

(iv) a **nullifies delete** rule asserts that the deletion of a tuple t' from R_j implies setting to null the subtuple $t[Y]$ in all the tuples t of R_i referencing t'.

Most commercial relational DBMSs provide either a procedural mechanisms for maintaining referential integrity constraints (e.g., Sybase 4.9, Ingres 6.3) or support a mix of procedural and declarative specifications for referential integrity constraints (e.g., Oracle 7, Sybase 10). Only single-target declarative referential integrity is provided by relational DBMSs. However, multi-target referential integrity constraints as well as exclusion constraints can be implemented using DBMS procedural mechanisms.

Commercial relational DBMSs support for specifying retrieval and update queries in their own dialects of SQL. Most relevant to our representation is retrieving data from multiple relations using *join* operations. Joining two relations, R_i and R_j, on join condition $A\theta A'$, where A is an attribute of R_i and A' is an attribute of R_j, involves finding tuples in R_i that have A values matching A' in tuples of R_j. For a (regular) join θ is '=' and its result includes only concatenated matching tuples. For an outer join θ is either '*=' (left outer join, where R_i is the outer relation and R_j is the inner relation) or '=*' (right outer join, where R_j is the outer relation and R_i is the inner relation). The result of outer join includes concatenated matching tuples plus tuples from the outer relation that have no matching tuples in the inner relation, padded with nulls.

While regular joins are associative, a left outerjoin followed by a regular join and a left outer join followed by a right outer join are not associative [Galindo-Jegaria & Rosenthal 92]. DBMSs such as Sybase 11 and Oracle 7 do not provide a way of specifying the order of executing outer joins in an SQL statement. Moreover, DBMSs such as Sybase 11 impose restrictions on combining outer joins (e.g., in Sybase, a relation cannot be involved in two left outer joins as both an inner and an outer relation).

2.2 Preliminary Mapping Steps

The mapping of an OPM schema into an relational schema is preceded by the following preliminary steps:

1. **Generate system metaclasses.** A system metaclass named `DATABASES` is generated to include all the local and foreign database definitions. `DATABASES` has three attributes: (i) a surrogate attribute called `_dbid`, (ii) an attribute called `_dbDescr` for database descriptions, and (iii) an attribute called `_dbName` for database names.

   If the OPM schema contains object classes, then generate system metaclasses `CLASSES` and `SCLASSES`. Metaclass `CLASSES` has six attributes: (i) a surrogate attribute called `_cid`, (ii) an attribute called `_alias` for class aliases, (iii) an attribute called `_classDescr` for class descriptions, (iv) an attribute called `_className` for class names, (v) an attribute called `_dbid` indicating for which database each class is defined, and (vi) an attribute called `_type` for class types (which can be `Object`, `Protocol`, `Derived Superclass`, or `Derived Subclass`). `SCLASSES` has two attributes, `_class` and `_subclass`, both associated with a value class consisting of `CLASSES`.

If the OPM schema contains protocol classes, then also generate system metaclass `SPROTOCOLS`, where `SPROTOCOLS` has two attributes, `_protocol` and `_subprotocol`, both associated with a value class consisting of `CLASSES`.

2. **Generate system classes.** If the OPM schema contains object classes and does not contain class `OBJECTS`, then generate system class `OBJECTS` with identifier attribute `Oid`; all object classes that are not subclasses of other object classes are specified as subclasses of `OBJECTS`. If the current schema supports permission modeling, then `OBJECTS` will have two additional attributes representing object *owner* and *status*.

   If the OPM schema contains protocol classes then generate system class `PROTOCOLS` with identifier attribute `Oid`; all protocol classes are specified as subclasses of `PROTOCOLS`. If the current schema supports permission model, then `PROTOCOLS` will have two additional attributes representing protocol *owner* and *status*.

3. **Generate system attributes.** Generate system attributes for protocol connections as follows:

   (a) System attribute `_inputFor`:
   If an object class O_i is involved in the value classes of input attributes A_{j_1} of protocol class P_{j_1}, ..., A_{j_m} of protocol class P_{j_m} ($m \geq 1$), then a system attribute `_inputFor` is automatically generated for O_i. Attribute `_inputFor` is associated with value class P_{j_1} or ... or P_{j_m}. `_inputFor` is associated with the `INPUT_FOR` delete rule (if any) specified for O_i; if this rule is not specified then `_inputFor` is associated with a (default) `NULLIFIES` delete rule. Attribute `_inputFor` is also associated with the following derivation: $!A_{j_1}\ [P_{j_1}]$ or ... or $!A_{j_m}\ [P_{j_m}]$.

   (b) System attribute `_outputOf`:
   If an object class O_i is involved in the value classes of output attributes A_{j_1} of protocol class P_{j_1}, ..., A_{j_m} of protocol class P_{j_m} ($m \geq 1$), then a system attribute `_outputOf` is automatically generated for O_i. Attribute `_outputOf` is associated with value class P_{j_1} or ... or P_{j_m}. `_outputOf` is associated with the `OUTPUT_OF` delete rule (if any) specified for O_i; if this rule is not specified then `_outputOf` is associated with a (default) `NULLIFIES` delete rule. Attribute `_outputOf` is also associated with the following derivation: $!A_{j_1}\ [P_{j_1}]$ or ... or $!A_{j_m}\ [P_{j_m}]$.

   (c) System attribute `_genProt`:
   If a protocol class P_i is involved in the protocol expansion of P_j, then a system attribute `_genProt` is automatically generated for P_i. The value class of `_genProt`

is P_j. _genProt is associated with the GENERIC_PROTOCOL delete rule (if any) specified for P_i; if this rule is not specified then _genProt is associated with a (default) NULLIFIES delete rule.

(d) System attribute _preProt:

If a protocol class P_i takes input from protocols P_{j_1}, ..., P_{j_m}, then a system attribute _preProt is automatically generated for P_i. The value class of _preProt is P_{j_1} or ... or P_{j_m}. _preProt is associated with the PREDECESSOR_PROTOCOLS delete rule (if any) specified for P_i; if this rule is not specified then _preProt is associated with a (default) NULLIFIES delete rule.

(e) System attribute _subProt:

If a protocol class P_i is expanded into subprotocols P_{j_1}, ..., P_{j_m}, then a system attribute _subProt is automatically generated for P_i. The value class of _subProt is P_{j_1} or ... or P_{j_m}. _subProt is associated with the SUBPROTOCOLS delete rule (if any) specified for P_i; if this rule is not specified then _subProt is associated with a (default) NULLIFIES delete rule. Attribute _subProt is also associated with the following derivation: !_genProt [P_{j_1}] or ... or !_genProt [P_{j_m}].

(f) System attribute _sucProt:

If a protocol class P_i is succeeded by protocols P_{j_1}, ..., P_{j_m}, then a system attribute _sucProt is automatically generated for P_i. The value class of _sucProt is P_{j_1} or ... or P_{j_m}. _sucProt is associated with the SUCCESSOR_PROTOCOLS delete rule (if any) specified for P_i; if this rule is not specified then _sucProt is associated with a (default) NULLIFIES delete rule. Attribute _sucProt is also associated with the following derivation: !_preProt [P_{j_1}] or ... or !_preProt [P_{j_m}].

4. Generate version system attributes.

If an abstract class O_i has (explicitly defined or inherited) versioned attributes, then the following version system attributes are automatically generated for O_i:

(a) non-versioned system attribute _defaultVersion

This is the default (or current) version number for instances in O_i. Attribute _defaultVersion is single-valued, can be null, and with value class INTEGER.

(b) non-versioned system attribute _nextVersion

This is the next version number for instances in O_i. Attribute _nextVersion is single-valued, not null, and with value class INTEGER.

(c) versioned system attribute _version

This represents the version number of an individual version. Attribute _version is single-valued, not null, and with value class INTEGER.

(d) versioned system attribute `_preVersion`

This represents the previous version from which this version is created. Attribute `_preVersion` is single-valued, can be null, and with value class `INTEGER`.

(e) versioned system attribute `_fromDate`

This records the date/time this version is generated. Attribute `_fromDate` is single-valued, not null, and with value class `DATETIME`.

(f) versioned system attribute `_deleteDate`

This records the date/time this version is logically deleted. Attribute `_deleteDate` is single-valued, can be null, and with value class `DATETIME`. If attribute value for `_deleteTime` is null, then this version in not marked as logically deleted.

5. Decide mapping order. Sort the object classes in the OPM schema into topological order: $S = \{ O_1, \ldots, O_n \}$, where for every pair O_i and O_j, $i < j$:

   if O_i and O_j are object classes, then O_i is not a subclass of O_j, and O_i is not a class derived from O_j.

6. Mark inverse attribute. If a simple attribute A associated with object or protocol class O_i is specified as the inverse of attribute B of class O_j, then **mark** A as "**inverse** of B of O_j" if B is not already marked, and

   (a) B is single-valued; or

   (b) A and B are both multi-valued, and $O_i > O_j$ in the topological order;

7. Mark arithmetic or aggregate derived attributes. If an attribute A is specified using a derivation rule involving an arithmetic or aggregate function expression, then **mark** A as "arithmetically/aggregate derived attribute".

8. Mark composition derived attributes. If an attribute A associated with an object or protocol class O_i is derived by attribute composition, then **mark** A as "composition derived attribute".

3 Mapping OPM Structures and Constraints

We describe in this section the mapping of OPM classes, attributes, and constraints into relations and constraints in relational databases.

3.1 Mapping Outline and Motivation

An OPM instance, x, can be perceived as consisting of a tuple of the form ($oid(x)$, $A_1 : A_1(x)$, ..., $A_k : A_k(x)$), where $oid(x)$ is the identifier of x, A_j, $1 \leq j \leq k$, is the name of an attribute of O_i, and $A_j(x)$ denotes the value of attribute A_j for x.

Definition 1 (Class Instances) *Let O_i be an OPM class and x be an instance of O_i (denoted $x \in O_i$). Then $x = (oid(x), val(x))$, where $oid(x)$ is the globally unique (internal) identifier of x and $val(x)$ is the value of x. $oid(x)$ takes values from a system-provided primitive value class of identifiers. $val(x)$ has the form ($A_1 : A_1(x)$, ..., $A_n : A_n(x)$), where A_j, $1 \leq j \leq n$, is the name of an attribute of O_i, and $A_j(x)$ denotes the value of attribute A_j for x and is defined as follows:*

1. *If A_j is a simple attribute associated with primitive value class V, then $A_j(x)$ consists of one element, a set of elements, or a list of elements from the domain of V.*
2. *If A_j is a simple abstract attribute associated with value class: O_1 or ... or O_m, then $A_j(x)$ consists of one element, a set of elements, or a list of elements from $\{ oid(y) \mid y \in O_k, 1 \leq k \leq m \}$.*
3. *If A_j = (A_{j_1}, ..., A_{j_n}), then $A_j(x)$ consists of a tuple, a set of tuples, or a list of tuples of the form [$A_{j_1} : A_{j_1}(x)$, ..., $A_{j_n} : A_{j_n}(x)$], where each component $A_{j_k}(x)$, $j_1 \leq j_k \leq j_n$, consists of one element of the form defined in 1 or 2 above.*

By definition, if y is a value from a primitive value class then $oid(y) = y$.

Definition 2 (Object Versions) *Let O be a versioned object class and x be an instance of O. Then $x = (oid(x), val(x), \{ (vid(x_i), vval(x_i) \})$, where $oid(x)$ is the globally unique object identifier of x, $val(x)$ is the non-versioned value of x, $vid(x_i)$ is a unique version identifier for version x_i of x, $vval(x_i)$ is a versioned value for version x_i of x. $oid(x)$ and $val(x)$ are defined as in Definition 1 above. $vid(x_i)$ takes values from a value class of version identifiers. $vval(x_i)$ has the form ($A_1 : A_1(x_i)$, ..., $A_n : A_n(x_i)$), where A_j, $1 \leq j \leq n$, is the name of a versioned attribute of O, and $A_j(x_i)$ denotes the value of attribute A_j for version x_i of x and is defined as follows:*

1. *If A_j is a simple primitive versioned attribute associated with value class V, then $A_j(x_i)$ consists of one element, a set of elements, or a list of elements from the domain of V.*

2. *If A_j is a simple abstract versioned attribute associated with value class O_1 or ... or O_m, then $A_j(x_i)$ consists of one element, a set of elements, or a list of elements from $\{$ $(oid(y), vid(y_p))$ $|$ $y \in O_k$, $1 \leq k \leq m$, and $vid(y_p)$ identifies version y_p of y $\}$.*

3. *If A_j = $(A_{j_1}, \ldots, A_{j_n})$, then $A_j(x_i)$ consists of a tuple, a set of tuples, or a list of tuples of the form $[A_{j_1} : A_{j_1}(x_i), \ldots, A_{j_n} : A_{j_n}(x_i)]$, where each component $A_{j_k}(x_i)$, $j_1 \leq j_k \leq j_n$, consists of one element of the form defined in 1 or 2 above.*

A version x_v of x is defined as $(oid(x), val(x), vid(x_v), vval(x_v))$.

Alternatively, an instance x can be perceived as consisting of a tuple of the form ($oid(x)$, $A_1 : A_1(x), \ldots, A_k : A_k(x)$), where A_j, $1 \leq j \leq k$, are single-valued non-versioned attributes, a tuple of the form ($oid(x)$, $A_{k+1} : A_{k+1}(x), \ldots, A_l : A_l(x)$), where A_j, $(k+1) \leq j \leq l$, are single-valued versioned attributes, and a set of pairs of the form $(oid(x), A_j : A_j(x))$, $l+1 \leq j \leq n$, where A_j is a set-valued or list-valued (versioned or non-versioned) attribute. Then the instances in a class O_i can be represented straightforwardly in a relational database as follows:

1. All single-valued non-versioned attributes of O_i are represented by attributes of a primary relation Rp_i that has an attribute representing object or protocol identifiers (oids), Oid_i. Oid_i is specified as the primary-key of Rp_i while attributes corresponding to identifier attributes of O_i are specified as an alternate key of Rp_i.

   For each single-valued abstract attribute A_j, if A_j is associated with a value class that includes class O_m then the relational attribute representing A_j, B_j is defined as a foreign-key referencing the primary relation representing O_m, Rp_m.

2. All single-valued versioned attributes of O_i are represented by attributes of a primary version relation Rv_i, which has a structure similar to Rp_i. System attribute `_defaultVersion#` is represented by an attribute of Rp_i; all other versioned system attributes are represented by attributes of Rv_i. The primary key for Rv_i is the pair of attributes representing Oid_i and version number v_i. Attribute Oid_i is specified as a foreign-key referencing Rp_i. The pair of attributes representing Oid_i and `_defaultVersion#` in Rp_i is specified as a foreign-key referencing Rv_i.

   For each single-valued versioned abstract attribute A_j, where A_j is associated with a value class that includes class O_m and A_j is represented by relational attribute B_j, if A_j references generic instances of O_m then B_j is defined as a foreign-key referencing the primary relation representing O_m, Rp_m; if A_j references specific versions of O_m then an additional attribute v_j is used for representing versions of O_m instances and the pair B_j and v_j is defined as a foreign-key referencing the primary version relation representing O_m, Rv_m.

3. An auxiliary relation Ra_{i_j} is employed for representing each set-valued or list-valued non-versioned attribute A_j of O_i. Relation Ra_{i_j} has three attributes, Aid, Oid_i, and B_j, where Aid represents a (local) attribute value identifier, Oid_i represents the oid for O_i, and B_j represents A_j; Aid is specified as the primary-key of Ra_{i_j}; attribute Oid_i in Ra_{i_j} is specified as a foreign-key referencing Oid_i in Rp_i.

   For each abstract attribute A_j, if A_j is associated with a value class that includes class O_m, then B_j is defined as a foreign-key referencing the primary relation representing O_m, Rp_m. If A_j is list-valued then an additional attribute $Order_j$ is used for representing the order of values in the list.

4. An auxiliary version relation Rav_{i_j} is employed for representing each set-valued or list-valued attribute A_j of O_i. Rav_{i_j} has a structure similar to the auxiliary relations employed for representing non-versioned attributes, but contain an additional attribute, v_i, representing the version number. The pair of attributes Oid_i and v_i are specified as a foreign-key referencing the primary version relation for O_i, Rv_i.

   For each set-valued or list-valued versioned abstract attribute A_j, where A_j is associated with a value class that includes class O_m and A_j is represented by relational attribute B_j, if A_j references generic instances of O_m then B_j is defined as a foreign-key referencing the primary relation representing O_m, Rp_m; if A_j references specific versions of O_m then an additional attribute v_j is used for representing versions of O_m instances and the pair B_j and v_j is defined as a foreign-key referencing the primary version relation representing O_m, Rv_m.

5. If O_i is specified as a subclass of O_k then attribute Oid_i in Rp_i is specified as a foreign-key referencing the primary relation representing O_k, Rp_k. If O_i and O_k have versioned attributes, then the pair of attributes Oid_i and v_i in the primary version relation representing O_i, Rv_i, is specified as a foreign-key referencing the primary version relation representing O_k, Rv_k.

   Note that if A_j of O_i is an attribute inherited from O_k then Ra_{i_j} is equal to the subset of the auxiliary relation representing A_j of O_k, Ra_{k_j}, for the oids appearing in Ra_{i_j}, and therefore is not represented since it would be redundant.

Derived attributes are not represented using relational attributes; their values are computed using SQL procedures representing retrieval methods associated with OPM classes (see next section). Note that inherited attributes of O_i that are inherited from a superclass O_j are not represented in the same way as non-inherited attributes of O_i. Instead, the primary and auxiliary relations of representing O_j are treated as auxiliary relations of O_i and are involved as such in the SQL procedures representing the retrieval method associated with O_i.

It can be shown easily that the relational representation above is in the highest normal form[1] and, together with additional relational constraints representing null, exclusion, and

[1] All (functional, multivalued) dependencies are key dependencies (see [Maier 83]).

inverse constraints, is sufficient for representing OPM classes.

Using normalized relations has the advantage of reduced data redundancy, comprehensibility, and simpler data integrity maintenance, but has the disadvantage of data fragmentation, that is, having data on one OPM instance scattered among multiple relations. Using unnormalized representations is also possible but is not desirable. For example, an alternative representation could use only one unnormalized relation for each OPM class; this relation can be obtained by left outer joining the primary relation with each of the auxiliary relations mentioned above. Using unnormalized relations has the advantage of having data on one OPM instance available in a single relation, but has the disadvantage of high data redundancy in a large and hard to comprehend relation, and complex data integrity maintenance. Furthermore, data integrity in an unnormalized relation needs to be expressed with additional, more complex constraints, such as general (non-key) functional dependency constraints, general null constraints, multivalued dependency constraints, and general inclusion dependency constraints [2]. These constraints are not supported directly by commercial relational DBMSs and therefore are very hard to maintain in such DBMSs.

An OPM derived subclass or superclass is mapped to a relational view with one single attribute representing the object identities (oids). Derivation and condition (if any) are translated into relational condition in the view definition. A derived class can have only derived attributes (except for oids). A derived attribute of a derived class is mapped as a derived attribute of a non-derived class.

If the current schema supports permission modeling, and a class O_i is *not* specified with `PERMISSION OFF`, then the *owner* and *status* attributes are de-normalized into each primary, version and auxiliary relations representing class O_i. The mapping of the *owner* attribute is the same as for single-valued abstract attributes described below (except that it is de-normalized into each relation representing O_i), and the mapping of the *status* attribute is the same as for single-valued primitive or controlled attributes described below (except that it is de-normalized into each relation representing O_i).

3.2 Mapping Details

Mapping an OPM schema into a relational schema involves using `surrogate` attributes. A surrogate attribute has a special data type, such as data type `identity` in Sybase 10, and has values that are automatically generated by the underlying DBMS.

[2]Multivalued dependencies are needed for expressing the independence of multi-valued attributes in an unnormalized relation. General inclusion dependencies are similar to referential integrity, but may involve non-primary key attributes in the referenced relations. See [Maier 83] for definitions of various relational dependencies.

3.2.1 Mapping Metaclasses

Metaclass `DATABASES` is represented using a relation also called `DATABASES`. Relation `DATABASES` has (i) an attribute called `_dbDescr` with data type `VARCHAR(255)` representing database descriptors (truncated to 255 characters); (ii) an attribute called `_dbName` with data type `VARCHAR(80)` representing foreign database names; and (iii) a surrogate attribute called `_dbid` representing database identifiers, where `_dbid` is specified as the primary-key, and the data type for `DBid` is `INTEGER`.

For each foreign database specified in the OPM schema, the following SQL query is generated to insert the database information into relation `DATABASES`:

`insert DATABASES values` (*desc*, *name*, *dbid*)

where *desc* is the database descriptor truncated to 255 characters, *name* is the database name, and *dbid* is an automatically generated database identity.

Metaclass `CLASSES` is represented using a relation also called `CLASSES`. This relation has the following attributes: (i) an attribute called `_alias` with data type `VARCAHR(80)` representing class aliases; (ii) a surrogate attribute called `_cid` representing class identifiers, where `_cid` is specified as the primary-key, and the data type for `_cid` is `INTEGER`; (iii) an attribute called `_classDescr` with data type `VARCAHR(255)` representing class descriptors; (iv) an attribute called `_className` with data type `VARCHAR(32)` representing class names; (v) an attribute called `_dbid` with data type `INTEGER` referencing attribute `_dbid` of relation `DATABASES`, and representing in which (local or foreign) database is each class defined; and (v) an attribute called `_type` with data type `varchar(20)` indicating the class types, which can be: `Object`, `Protocol`, `Foreign`, `Derived Subclass`, or `Derived Superclass`.

The content of the relation `CLASSES` are generated as follows: For each object (or protocol) class O_i defined in the schema, an SQL insertion statement

`insert CLASSES values` (*alias*, *id*, *desc*, *name*, *dbid*, *type*)

is generated, where *id* is an automatically generated class identifier for O_i, *alias* is the provided class alias for O_i (if any), *desc* is the quoted class description for O_i (truncated to 255 characters), *name* is the quoted class name for O_i, *dbid* is the id for the database in which O_i is defined, and *type* is `''Object''` (if O_i is an object class), `''Protocol''` (if O_i is a protocol class), `''Foreign''` (if O_i is a foreign class), `''Derived Subclass''` (if O_i is a derived subclass), or `''Derived Superclass''` (if O_i is a derived superclass).

Metaclass `SCLASSES` is represented by a relation `SCLASSES` with two attributes `_class` and `_subclass`. Both attributes are with `INTEGER` data type. The contents of `SCLASSES` are generated as follows: for each object class O_i and its immediate subclass O_s, an SQL statement

`insert SCLASSES values` (id_i, id_s)

is generated, where id_i and id_s are class identifiers for O_i and O_s as in the `CLASSES` relation, respectively.

Metaclass `SPROTOCOLS` is represented by a relation `SPROTOCOLS` with two attributes `_protocol` and `_subprotocol`. Both attributes are with `INTEGER` data type. The contents of `SPROTOCOLS` are generated as follows: for each protocol class P_i and its immediate subprotocol P_s, an SQL statement

`insert SPROTOCOLS values` (id_i, id_s)

is generated, where id_i and id_s are class identifiers for P_i and P_s as in the `CLASSES` relation, respectively.

3.2.2 Mapping Named Primitive and Controlled Value Classes

Named primitive value classes are represented using relational user specified domains. There are two types of controlled value classes in OPM: (i) controlled value class of enumerated atomic values possibly with codes and/or value descriptions, and (ii) controlled value class of value ranges.

A controlled value class, V_i, consisting of value ranges or values without codes or descriptions is represented using range constraints associated with the domain of the relational attribute representing the attribute associated with V_i. For example, controlled value class `RANK` is mapped into a domain constraint with 3 values: "Professor", "Associate Professor" and "Assistant Professor" (see figure 2).

A controlled value class, V_i, consisting of values associated with codes and/or value descriptions, is represented using an auxiliary relation, R_v, where attributes of R_v represent the values, codes, and descriptions, respectively, and a range constraint is associated with the domain of the relational attribute representing the values. The attribute representing codes are defined as the primary key for R_v, and the attribute representing values are defined as alternate key. Relation R_v also has an additional attribute Aid representing local value identity. Furthermore, for each attribute A associated with V_i, the relational attribute representing A is specified as a foreign-key referencing R_v.

3.2.3 Mapping Classes

An OPM (non-derived) class, O_i, is mapped as followed:

1. if O_i is system class `OBJECTS` or system class `PROTOCOLS` then it is represented using a relation called `OBJECTS` or `PROTOCOLS`, respectively, associated with a surrogate attribute called `_oid`; `_oid` is specified as the primary-key;

2. if O_i is an object class, then it is represented using a primary relation Rp_i, that has an attribute representing oids, Oid_i;

(a) if O_i is not specified as a subclass of another object class, then attribute Oid_i in the primary relation representing O_i is specified as a foreign-key referencing `_oid` in relation `OBJECTS`;

(b) if O_i is specified as a subclass of another object class, O_j, then attribute Oid_i in the primary relation representing O_i is specified as a foreign-key referencing the primary relation representing O_j; if the subclass relationship is exclusive (ISA*) then an exclusion constraint involving Oid_i and all primary keys of the primary relations representing the other subclasses of O_j is also specified;

3. if O_i is a protocol class, then it is represented using a primary relation Rp_i, that has an attribute representing oids, Oid_i; Oid_i is specified as a foreign-key referencing `_oid` in relation `PROTOCOLS`.
4. if O_i is versioned, then an additional primary version relation Rv_i is also created for O_i. Relation Rv_i has an attribute representing oids, Oid_i, and an attribute representation version numbers, v_i; Oid_i is specified as a foreign-key referencing the primary relation representing O_i; the pair of attributes Oid_i and v_i is specified as the primary key of Rv_i; the primary relation representing O_i, Rp_i, is extended with an additional attribute $\_defaultVersion_i$ representing the default version of O_i, and an additional attribute `_nextVersion` representing the next version number. The pair of attributes Oid_i and $\_defaultVersion_i$ in Rp_i is specified as a foreign key referencing Rv_i.

3.2.4 Mapping Attributes

We now describe the details regarding the relational representation of non-derived OPM attributes. Depending on its complexity, a simple non-inherited attribute A of an OPM class O_i represented by primary relation Rp_i is mapped as follows: (Attribute A does not allow null values if its minimal cardinality is greater than 0.)

1. if A is not versioned, then

   (a) if A is primitive, then

      i. if A is single-valued, then A is represented by an attribute B of Rp_i;

      ii. if A is set-valued or list-valued, then A is represented by attribute B of *auxiliary* relation $Ra_A(Aid,\ Oid_i,\ B)$, where Aid is a surrogate attribute representing a (local) attribute value identifier, B is not allowed to have null values, the primary-key consists of Aid, and Oid_i is specified as a foreign-key referencing Rp_i and is associated with a *cascades* delete rule; if A is list-valued, then an additional attribute $Order_A$ is also generated;

   (b) if A is abstract and associated with a value class consisting of OPM classes O_{j1}, …, O_{jn} $(n \geq 1)$, where O_{jk} is represented by primary relation Rp_{jk} $(1 \leq k \leq n)$, then

i. if A is single-valued then A is represented by a foreign-key attribute B of Rp_i referencing Rp_{j_1}, ..., Rp_{j_n}, and associated with the delete rule specified for A with respect to O_{j_1}, ..., O_{j_n};

ii. if A is set-valued or list-valued, then A is represented by attribute B of *auxiliary* relation $Ra_A(Aid,\ Oid_i,\ B)$, where Aid is a surrogate attribute representing a (local) attribute value identifier, B is not allowed to have null values, the primary-key consists of Aid, Oid_i is specified as a foreign-key referencing Rp_i and is associated with a *cascades* delete rule, and B is specified as a foreign-key referencing Rp_{j_1}, ..., Rp_{j_n} and is associated with the delete rule specified for A with respect to O_{j_1}, ..., O_{j_n}; if A is list-valued, then an additional attribute $Order_A$ is generated;

2. if A is versioned, then

   (a) if A is primitive, then

   i. if A is single-valued, then A is represented by an attribute B of Rv_i;

   ii. if A is set-valued or list-valued, then A is represented by attribute B of *auxiliary* version relation $Rav_A(Aid,\ Oid_i,\ v_i,\ B)$, where Aid is a surrogate attribute representing a (local) attribute value identifier, B is not allowed to have null values, the primary-key consists of Aid; $(Oid_i,\ v_i)$ is specified as a foreign-key referencing Rv_i and is associated with a *cascades* delete rule; if A is list-valued, then an additional attribute $Order_A$ is also generated;

   (b) if A is abstract and has a generic reference to a value class consisting of OPM classes O_{j_1}, ..., O_{j_n} $(n \geq 1)$, where O_{j_k} is represented by primary relation Rp_{j_k} $(1 \leq k \leq n)$, then

   i. if A is single-valued then A is represented by a foreign-key attribute B of Rp_i referencing Rp_{j_1}, ..., Rp_{j_n}, and associated with the delete rule specified for A with respect to O_{j_1}, ..., O_{j_n};

   ii. if A is set-valued or list-valued, then A is represented by attribute B of *auxiliary* relation $Ra_A(Aid,\ Oid_i,\ v_i,\ B)$, where Aid is a surrogate attribute representing a (local) attribute value identifier, B is not allowed to have null values, the primary-key consists of Aid, and B is specified as a foreign-key referencing Rp_{j_1}, ..., Rp_{j_n} and is associated with the delete rule specified for A with respect to O_{j_1}, ..., O_{j_n}; $(Oid_i,\ v_i)$ is specified as a foreign-key referencing Rv_i and is associated with a *cascades* delete rule; if A is list-valued, then an additional attribute $Order_A$ is also generated;

   (c) if A is abstract and has a specific reference to a value class consisting of OPM classes O_{j_1}, ..., O_{j_n} $(n \geq 1)$, where O_{j_k} is represented by primary relation Rp_{j_k} $(1 \leq k \leq n)$ and primary version relation Rv_{j_k} $(1 \leq k \leq n)$, then

i. if A is single-valued then A is represented by a pair of attributes B and _vB, where (B, _vB) is specified as a foreign-key referencing primary version relations Rv_{j_1}, ..., Rv_{j_n} and is associated with the delete rule specified for A with respect to O_{j_1}, ..., O_{j_n};

ii. if A is set-valued or list-valued, then A is represented by attributes B and _vB of *auxiliary* relation $Ra_A(Aid, Oid_i, v_i, B,$ _v$B)$, where Aid is a surrogate attribute representing a (local) attribute value identifier, B is not allowed to have null values, the primary-key consists of Aid, and (Oid_i, v_i) is specified as a foreign-key referencing Rv_i and is associated with a *cascades* delete rule; (B, _vB) is specified as a foreign-key referencing primary version relations Rv_{j_1}, ..., Rv_{j_n}, and is associated with the delete rule specified for A with respect to O_{j_1}, ..., O_{j_n}; if A is list-valued, then an additional attribute $Order_A$ is also generated.

Tuple attributes are mapped in a way similar to simple attributes.

If two attributes, A of O_i and A' of O_j, are specified as inverses of each other, then only one auxiliary relation and/or relational attribute are generated for representing both attributes [3]. Suppose that attribute A' of O_j is represented using an attribute of the primary relation or an auxiliary relation representing O_j. Then this relation is also considered as an auxiliary relation for O_i and is used in the SQL procedure representing the retrieval method associated with O_i.

If attribute A of O_i has an attribute default value $Default_A$ specified in the OPM schema definition, then the relational attribute representing A, B, is also associated with a default value $Default_B$:

1. If A is associated with a controlled value class C with codes, then $Default_B$ is the corresponding code value in C for controlled value $Default_A$.

2. Otherwise, $Default_B$ is the same as $Default_A$.

The `(min, max)` cardinality constraint associated with an attribute A is not mapped to constructs in relational schema definition. Instead, the cardinality constraint is implemented as a checking procedure (e.g., Sybase stored procedure) that should be envoked during each update to attribute A.

Example 1. Object class `FACULTY` shown in figure 1 is represented by primary relation `FACULTY`, primary version relation `_vFACULTY`, and auxiliary relations `FACULTY_Teaches` and `PERSON` (see figure 2):

[3] When choosing the inverse attribute that is directly represented, a component attribute is preferred over a simple attribute, an attribute associated with a value class consisting of several OPM classes is preferred over an attribute associated with a value class consisting of a single OPM class, and a single-valued attribute is preferred over a set or list valued attribute.

1. The oids are represented by attribute `_oid` of relation `FACULTY`, where `_oid` is specified as a primary key. Single-valued primitive attributes `Rank` is represented by attribute `Rank` of `FACULTY`. Single-valued abstract attribute `Department` is represented by attribute `Department` of `FACULTY`, where `Department` is specified as a foreign-key referencing relation `DEPARTMENT`. Since class `FACULTY` has versioned attributes, two additional attribute `_defaultVersion` and `_nextVersion` are also generated, and foreign-key (`_oid`, `_defaultVersion`) is specified as referencing relation `_vFACULTY`.
2. The primary version relation `_vFACULTY` has primary-key attributes `_oid` and `_version`, and other version system attributes `_preVersion`, `_fromDate` and `_deleteDate`.
3. Set-valued versioned composite attribute `Teaches(Course, InDepartment)` is represented by attributes `Course` and `InDepartment` of auxiliary relation `FACULTY_Teaches`, where `Course` is specified as a foreign-key referencing `COURSE` and `InDepartment` is specified as a foreign-key referencing `DEPARTMENT`. `FACULTY_Teaches` also has primary-key attribute `_aid` and foreign-key (`_oid`, `_version`) referencing relation `_vFACULTY`. The cascade delete rule associated with this foreign-key is implemented using a trigger procedure implementing cascade deletions from `FACULTY` to `FACULTY_Teaches` (not shown in figure 2):
4. The isa relationship from `FACULTY` to `PERSON` is represented by specifying `_oid` in `FACULTY` relation as a foreign-key referencing `PERSON` relation. Inherited attributes of `FACULTY` from `PERSON` are not represented as attributes of relation `FACULTY`; instead, they are represented as attributes of relation `PERSON`.

□

3.3 Mapping Derived Classes

Let class O_s be a specialization derived class with derivation: subclass of O_1, ..., O_m ($m \geq 1$) without condition. Suppose classes O_1, ..., O_m are represented by primary tables or views R_1, ..., R_m, respectively.

O_s is represented by a relational view V_s with the following definition:

create view V_s (_oid)
as select _oid from R_1
where _oid in (select _oid from R_2)
...
and _oid in (select _oid from R_m)

If O_s is associated with a derivation condition: $cond_1$ and ... $cond_n$ ($n \geq 1$), then each subcondition $cond_i$ ($1 \leq i \leq n$) is translated to a corresponding relational condition $cond'_i$, and added to the above view definition. The procedure for translating OPM subcondition $cond_i$ to relational condition $cond'_i$ is described immediately below.

```
create table PERSON (
        _oid numeric(10, 0) not null,
        Age AGE null,
        Name varchar(50) null,
        SSN int not null,
        primary key (_oid),
        foreign key (_oid) references OBJECTS )
create table FACULTY (
        _defaultVersion int null,
        _nextVersion int null,
        _oid numeric(10, 0) not null,
        Department numeric(10, 0) null,
        Rank RANK null,
        primary key (_oid),
        foreign key (_oid) references PERSON,
        foreign key (_oid, _defaultVersion) references _vFACULTY,
        foreign key (Department) references DEPARTMENT )
create table _vFACULTY (
        _deleteDate datetime null,
        _fromDate datetime not null,
        _oid numeric(10, 0) not null,
        _preVersion int null,
        _version int not null,
        primary key (_oid, _version#),
        foreign key (Oid) references FACULTY )
create table FACULTY_Teaches (
        _aid numeric(10, 0) identity,
        _oid numeric(10, 0) not null,
        _version int not null,
        Course numeric(10, 0) not null,
        InDepartment numeric(10, 0) not null,
        primary key (_aid),
        foreign key (_oid, _version) references _vFACULTY,
        foreign key (Course) references COURSE,
        foreign key (InDepartment) references DEPARTMENT )
```

Figure 2: Sybase Tables and Foreign Keys Representing Object Class FACULTY

We assume below that class O_j and O_k are in { O_1, ..., O_m }, and are represented by primary relations or views R_j and R_k, respectively. A is a non-derived attribute of O_j, and is represented by a relational attribute A' of R_A (R_A can be R_j). B is a non-derived attribute of O_k, and is represented by a relational attribute B' of R_B (R_B can be R_k).

1. If $cond_i$ is: `NOT IN` O_l

   Then $cond'_i$ is: `not in ( select _oid from` R_l `)`, where R_l is the primary relation or view representing class O_l. This subcondition $cond'_i$ is attached to the end of view definition V_s.

2. If $cond_i$ is: [O_j] A `IS NULL/IS NOT NULL`

   (a) if A is single-valued and non-versioned (and therefore is represented by a relational attribute in R_j), then $cond'_i$ is: A' `is null/is not null`; condition $cond'_i$ is attached to the select subquery associated with R_j;

   (b) otherwise, $cond'_i$ is: `not in/in ( select _oid from` R_A `where _oid is not null )`; condition $cond'_i$ is attached to the end of view definition V_s.

3. If $cond_i$ is: [O_j] A `MATCH/NOT MATCH` s, where s is a character string

   (a) if A is single-valued and non-versioned (and therefore is represented by a relational attribute in R_j), then $cond'_i$ is: A' `like/not like` s; condition $cond'_i$ is attached to the select subquery associated with R_j;

   (b) otherwise, $cond'_i$ is: `in ( select _oid from` R_A `where` A' `like/not like` s `)`; condition $cond'_i$ is attached to the end of view definition V_s.

4. If $cond_i$ is: [O_j] A `IN/NOT IN` S, where S is set of numbers or strings { s_1, ..., s_t }

   (a) if A is single-valued and non-versioned (and therefore is represented by a relational attribute in R_j), then $cond'_i$ is: A' `in/not in (` s_1, ..., s_t `)`; condition $cond'_i$ is attached to the select subquery associated with R_j;

   (b) otherwise, $cond'_i$ is: `_oid not in/in ( select _oid from` R_A `where` A' `not in (` s_1, ..., s_t `) )`; condition $cond'_i$ is attached to the end of view definition V_s.

5. If $cond_i$ is: [O_j] A `IN/NOT IN` O_l, where O_l is an OPM class represented by a relational table or view R_l

   (a) if A is single-valued and non-versioned (and therefore is represented by a relational attribute in R_j), then $cond'_i$ is: A' `in/not in ( select _oid from` R_l `)`; condition $cond'_i$ is attached to the select subquery associated with R_j;

(b) otherwise, $cond'_i$ is: `_oid not in/in ( select _oid from` R_A `where` A' `not in ( select _oid from` R_l `)`; condition $cond'_i$ is attached to the end of view definition V_s.

6. If $cond_i$ is: [O_j] A `IN/NOT IN` [O_k] B

   Then $cond'_i$ is: `_oid not in/in ( select _oid from` R_A `where` A' `not in ( select _oid from` R_B `where _oid =` R_A`._oid ) )` condition $cond'_i$ is attached to the end of view definition V_s.

7. If $cond_i$ is: [O_j] A op n, where n is a number, a string, a set of numbers or a set of strings, and $op \in \{ =, \neq, =$ `ANY`, $\neq$ `ANY`, $=$ `ALL`, $\neq$ `ALL` $\}$ if n is of string type, $op \in \{ =, \neq, <, \leq, >, \geq, =$ `ANY`, $\neq$ `ANY`, $<$ `ANY`, $\leq$ `ANY`, $>$ `ANY`, $\geq$ `ANY`, $=$ `ALL`, $\neq$ `ALL` $<$ `ALL`, $\leq$ `ALL`, $>$ `ALL`, $\geq$ `ALL` $\}$ if n is of numeric type.

   (a) if A is single-valued and non-versioned (and therefore is represented by a relational attribute in R_j), then $cond'_i$ is: A' op n;[4] condition $cond'_i$ is attached to the select subquery associated with R_j;

   (b) otherwise, $cond'_i$ is: `_oid in ( select _oid from` R_A `where` A' op n `)`;[4]condition $cond'_i$ is attached to the end of view definition V_s.

8. If $cond_i$ is: [O_j] A op [O_k] B, where $op \in \{ =, \neq, <, \leq, >, \geq, =$ `ANY`, $\neq$ `ANY`, $<$ `ANY`, $\leq$ `ANY`, $>$ `ANY`, $\geq$ `ANY`, $=$ `ALL`, $\neq$ `ALL` $<$ `ALL`, $\leq$ `ALL`, $>$ `ALL`, $\geq$ `ALL` $\}$

   (a) if A is single-valued and non-versioned (and therefore is represented by a relational attribute in R_j), then $cond'_i$ is: A' op `( select` B' `from` R_B `where` R_B`._oid =` R_j`._oid )`;[5] condition $cond'_i$ is attached to the select subquery associated with R_j;

   (b) otherwise, $cond'_i$ is: `_oid in ( select _oid from` R_A `where` A' op `( select` B' `from` R_B `where` R_B`._oid =` R_A`._oid ) )`;[5]condition $cond'_i$ is attached to the end of view definition V_s.

Example 2. Derived object class `PROFESSOR` shown in figure 1 is represented by relational view `PROFESSOR` with the following definition:

```
create view PROFESSOR ( _oid )
as select _oid from FACULTY
where rank = "Professor"
```

[4]If n is a set, then the OPM set symbols {} have to be replaced by the SQL set symbols () in $cond'_i$. Also, comparators $=, \neq, <$, etc. will be replaced by $=$ `ANY`, $\neq$ `ANY`, $<$ `ANY`, etc., respectively.

[5]If op is $=, \neq, <$, etc., then op will be replaced by $=$ `ANY`, $\neq$ `ANY`, $<$ `ANY`, etc., respectively.

Let class O_g be a generalization derived class with derivation: O_1 or ... or O_m ($m > 1$). Suppose classes O_1, ..., O_m are represented by primary tables or views R_1, ..., R_m, respectively.

If classes O_1, ..., O_m has common superclasses in the OPM schema, and let O_{lcs} be the *least common superclass* of O_1, ..., O_m, then O_g is represented by a relational view V_g with the following definition:

```
create view Vg ( _oid )
as select _oid from Rlcs
where _oid in ( select _oid from R1 )
         ...
or _oid in ( select _oid from Rm )
```

where R_{lcs} is the relational table or view representing O_{lcs}.

If O_1, ..., O_m have no common superclasses (except for `OBJECTS`), then O_g is represented by a relational view V_g defined as follows:

```
create view Vg ( _oid )
as select _oid from OBJECTS
where _oid in ( select _oid from R1 )
         ...
or _oid in ( select _oid from Rm )
```

4 Retrieval and Data Manipulation Methods

In this section, we examine the problems involved in representing retrieval methods in a relational database. Update methods are described in next section.

4.1 Representing Methods

Representing an OPM schema in a relational database entails representing not only class definitions using relations and constraints, but also retrieval and update methods using relational DBMS specifications. Since data on instances of an OPM class in a relational database is scattered among several relations, retrieval and update methods usually involve multiple SQL queries. Suppose that an OPM class, O_i, is represented using a *primary* relation, a primary version relation, and several *auxiliary* (version or non-version) relations, as discussed in the previous section. Then the retrieval method associated with O_i consists of left outer joining the primary relation representing O_i with the primary version relation and auxiliary relations representing O_i as well as with the primary relations representing the OPM classes involved in the value classes of the attributes of O_i. Furthermore, the retrieval method associated with O_i must also include computing the values of the derived attributes of O_i.

An OPM derived class O_i is represented by a view with one single attribute `_oid`. Therefore, object identities for all objects in O_i can be retrieved easily from the view representing O_i. O_i can only have derived attributes (except for `_oid`). Retrieval methods are associated with derived attributes of O_i as derived attributes of non-derived classes.

A retrieval method for an OPM class O_i consists of an SQL procedure that is constructed from SQL subqueries representing retrieval methods for each attribute of O_i. We assume below that OPM class O_i is represented by primary relation or view (if O_i is a derived class) Rp_i with primary-key attribute `_oid` (and primary version relation Rv_i with primary-key attributes `_oid` and `_version`).

4.2 Retrieval Methods for Non-Derived Attributes

Let non-derived local attribute A of OPM class O_i be represented by relational attribute B. (In this case, O_i cannot be a derived class.) As described in the previous section, B is an attribute of Rp_i or Rv_i if A is single-valued; otherwise B is an attribute of an auxiliary relation $Ra_A(Oid_i, B, \ldots)$.

A retrieval method for attribute A, `retrieve`(oid, $[v]$), with an object identity and an optional version number is specified by an SQL subquery SQL_A as follows:

1. if A is not versioned
   (a) if A is single-valued

i. if A is primitive, then SQL_A is:
`select` B `from` Rp_i `where _oid` $= oid$;

ii. if A is abstract with value class O_j, and O_j does not have REP, then SQL_A is:
`select` B `from` Rp_i `where _oid` $= oid$;

iii. if A is abstract with value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
`select` $D_1, \ldots, D_k$ `from` Rp_i, Rp_j
`where` Rp_i`._oid` $= oid$ `and` $B = Rp_j$`._oid`;

(b) if A is a set-valued

i. if A is primitive, then SQL_A is:
`select` B `from` Ra_A `where` Ra_A`._oid` $= oid$;

ii. if A is abstract with value class O_j, and O_j does not have REP, then SQL_A is:
`select` B `from` Ra_A `where` Ra_A`._oid` $= oid$;

iii. if A is abstract with value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
`select` $D_1, \ldots, D_k$ `from` Ra_A, Rp_j
`where` Ra_A`._oid` $= oid$ `and` $B = Rp_j$`._oid`;

(c) if A is a list-valued

i. if A is primitive, then SQL_A is:
`select` B `from` Ra_A `where` Ra_A`._oid` $= oid$
`order by` $Order_A$;

ii. if A is abstract with value class O_j, and O_j does not have REP, then SQL_A is:
`select` B `from` Ra_A `where` Ra_A`._oid` $= oid$
`order by` $Order_A$;

iii. if A is abstract with value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
`select` $D_1, \ldots, D_k$ `from` Ra_A, Rp_j
`where` Ra_A`._oid` $= oid$ `and` $B = Rp_j$`._oid`
`order by` $Order_A$;

(d) If A is an abstract attribute of O_i specified as inverse of abstract attribute A' of O_j

i. if A' is represented by relational attribute B' of primary relation $R_j(Oid_j, B', \ldots)$

A. if O_j does not have REP attributes, then the SQL_A is:
`select` $A =$ `_oid from` R_j `where` $B' = oid$;

B. if O_j has REP attributes $D_1, \ldots, D_k$, then the SQL_A is:
`select` $D_1, \ldots, D_k$ `from` R_j `where` $B' = oid$;

ii. if A' is represented by relational attribute B' of auxiliary relation $R_{A'}(Oid_j, B', \ldots)$,

A. if O_j does not have REP attributes, then the SQL_A is:
`select` A= `_oid from` $R_{A'}$ `where` $B' = oid$;

B. if O_j has REP attributes $D_1, \ldots, D_k$, then the SQL_A is:
`select` $D_1, \ldots, D_k$ `from` R_j, $R_{A'}$
`where` R_j`._oid` = $R_{A'}$`.Oid and` $B' = oid$;

2. If A is versioned and v is specified

(a) if A is single-valued

i. if A is primitive, then SQL_A is:
`select` B `from` Rv_i `where _oid` = oid
`and _version` = v;

ii. if A is abstract with *generic* reference and value class O_j, and O_j does not have REP, then SQL_A is:
`select` B `from` Rv_i `where _oid` = oid
`and _version` = v;

iii. if A is abstract with *generic* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
`select` $D_1, \ldots, D_k$ `from` Rv_i, Rp_j
`where` Rv_i`._oid` = oid `and` Rv_i`._version` = v
`and` B = Rp_j`._oid`;

iv. if A is abstract with *specific* reference and value class O_j, and O_j does not have REP, then SQL_A is:
`select` B, B`_v# from` Rv_i
`where _oid` = oid `and _version` = v;

v. if A is abstract with *specific* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
`select` $D_1, \ldots, D_k$, `_b`B `from` Rv_i, Rp_j
`where` Rp_i`._oid` = oid `and` Rp_i`._version` = v
`and` B = Rp_j`._oid`;

(b) if A is a set-valued

i. if A is primitive, then SQL_A is:
`select` B `from` Ra_A `where` Ra_A`._oid` = oid
`and _version` = v;

ii. if A is abstract with *generic* reference and value class O_j, and O_j does not have REP, then SQL_A is:
`select` B `from` Ra_A `where` Ra_A`._oid` $= oid$
`and _version` $= v$;

iii. if A is abstract with *generic* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
`select` $D_1, \ldots, D_k$ `from` Ra_A, Rp_j
`where` Ra_A`._oid` $= oid$ `and` Ra_A`._version` $= v$
`and` $B = Rp_j$`._oid`;

iv. if A is abstract with *specific* reference and value class O_j, and O_j does not have REP, then SQL_A is:
`select` B, `_v`B `from` Ra_A `where` Ra_A`._oid` $= oid$
`and _version` $= v$;

v. if A is abstract with *specific* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
`select` $D_1, \ldots, D_k$, `_v`B `from` Ra_A, Rp_j
`where` Ra_A`._oid` $= oid$ `and` Ra_A`._version` $= v$
`and` $B = Rp_j$`._oid`;

(c) if A is a list-valued

i. if A is primitive, then SQL_A is:
`select` B `from` Ra_A
`where` Ra_A`._oid` $= oid$ `and _version` $= v$
`order by` $Order_A$;

ii. if A is abstract with *generic* reference and value class O_j, and O_j does not have REP, then SQL_A is:
`select` B `from` Ra_A `where` Ra_A`._oid` $= oid$ `and _version` $= v$
`order by` $Order_A$;

iii. if A is abstract with *generic* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
`select` $D_1, \ldots, D_k$ `from` Ra_A, Rp_j
`where` Ra_A`._oid` $= oid$ `and` Ra_A`._version` $= v$
`and` $B = Rp_j$`._oid`
`order by` $Order_A$;

iv. if A is abstract with *specific* reference and value class O_j, and O_j does not have REP, then SQL_A is:
`select` B, `_v`B `from` Ra_A `where` Ra_A`._oid` $= oid$ `and _version` $= v$
`order by` $Order_A$;

v. if A is abstract with *specific* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:

select D_1, ..., D_k, _vB from Ra_A, Rp_j
where Ra_A._oid = oid and Ra_A._version = v
and $B = Rp_j$._oid
order by $Order_A$;

3. If A is versioned and v is not specified (i.e., using default)

   (a) if A is single-valued

      i. if A is primitive, then SQL_A is:
      select $Rv_i.B$ from Rp_i, Rv_i
      where Rp_i._oid = oid and Rv_i._oid = oid
      and Rp_i._defaultVersion = Rv_i._version;

      ii. if A is abstract with *generic* reference and value class O_j, and O_j does not have REP, then SQL_A is:
      select $Rv_i.B$ from Rp_i, Rv_i
      where Rp_i._oid = oid and Rv_i._oid = oid
      and Rp_i._defaultVersion = Rv_i._version;

      iii. if A is abstract with *generic* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) D_1, ..., D_k, then SQL_A is:
      select D_1, ..., D_k from Rp_i, Rv_i, Rp_j
      where Rp_i._oid = oid and Rv_i._oid = oid
      and Rp_i._defaultVersion = Rv_i._version
      and $B = Rp_j$._oid;

      iv. if A is abstract with *specific* reference and value class O_j, and O_j does not have REP, then SQL_A is:
      select B, _vB from Rp_i, Rv_i
      where Rp_i._oid = oid and Rv_i._oid = oid
      and Rp_i._defaultVersion = Rv_i._version;

      v. if A is abstract with *specific* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) D_1, ..., D_k, then SQL_A is:
      select D_1, ..., D_k, _vB from Rp_i, Rv_i, Rp_j
      where Rp_i._oid = oid and Rv_i._oid = oid
      and Rp_i._defaultVersion = Rv_i._version
      and $B = Rp_j$._oid;

   (b) if A is a set-valued

      i. if A is primitive, then SQL_A is:
      select B from Rp_i, Ra_A
      where Rp_i._oid = oid and Ra_A._oid = oid
      and Rp_i._defaultVersion = Ra_A._version;

ii. if A is abstract with *generic* reference and value class O_j, and O_j does not have REP, then SQL_A is:
select B from Rp_i, Ra_A where Rp_i._oid $= oid$ and Ra_A._oid $= oid$
and Rp_i._defaultVersion $= Ra_A$._version;

iii. if A is abstract with *generic* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
select $D_1, \ldots, D_k$ from Rp_i, Ra_A, Rp_j
where Rp_i._oid $= oid$ and Ra_A._oid $= oid$
and Rp_i._defaultVersion $= Ra_A$._version
and $B = Rp_j$._oid;

iv. if A is abstract with *specific* reference and value class O_j, and O_j does not have REP, then SQL_A is:
select B, B_v# from Rp_i, Ra_A
where Rp_i._oid $= oid$ and Ra_A._oid $= oid$
and Rp_i._defaultVersion $= Rv_i$._version;

v. if A is abstract with *specific* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
select $D_1, \ldots, D_k$, _vB from Rp_i, Ra_A, Rp_j
where Rp_i._oid $= oid$ and Ra_A._oid $= oid$
and Rp_i._defaultVersion $= Ra_A$._version
and $B = Rp_j$._oid;

(c) if A is a list-valued

i. if A is primitive, then SQL_A is:
select B from Rp_i, Ra_A
where Rp_i._oid $= oid$ and Ra_A._oid $= oid$
and Rp_i._defaultVersion $= Ra_A$._version
order by $Order_A$;

ii. if A is abstract with *generic* reference and value class O_j, and O_j does not have REP, then SQL_A is:
select B from Rp_i, Ra_A where Rp_i._oid $= oid$ and Ra_A._oid $= oid$
and Rp_i._defaultVersion $= Ra_A$._version
order by $Order_A$;

iii. if A is abstract with *generic* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
select $D_1, \ldots, D_k$ from Rp_i, Ra_A, Rp_j
where Rp_i._oid $= oid$ and Ra_A._oid $= oid$
and Rp_i._defaultVersion $= Ra_A$._version
and $B = Rp_j$._oid
order by $Order_A$;

iv. if A is abstract with *specific* reference and value class O_j, and O_j does not have REP, then SQL_A is:
`select` B, `_v`B `from` Rp_i, Ra_A `where` Rp_i`._oid` $= oid$ `and` Ra_A`._oid` $= oid$
`and` Rp_i`._defaultVersion` $=$ Ra_A`._version`
`order by` $Order_A$;

v. if A is abstract with *specific* reference and value class O_j, and REP attributes for O_j is (represented by relational attributes) $D_1, \ldots, D_k$, then SQL_A is:
`select` $D_1, \ldots, D_k$, `_v`B `from` Rp_i, Ra_A, Rp_j
`where` Rp_i`._oid` $= oid$ `and` Ra_A`._oid` $= oid$
`and` Rp_i`._defaultVersion` $=$ Ra_A`._version`
`and` $B = Rp_j$`._oid`
`order by` $Order_A$;

If attribute A in class O_i is inherited from superclass O_s, then retrieval method for A is the same as local attributes except that Rp_i is replaced by primary relation for O_s, Rp_s, and Rv_i is replaced by primary version relation for O_s, Rv_s.

4.3 Retrieval Methods for Arithmetic or Aggregate Function Derived Attributes

Let A be a local derived attribute of a non-derived class O_i. A retrieval method for attribute A that retrieves values of A together with associated oids consists of an SQL subquery, SQL_A, specified as follows:

1. If A is an arithmetic derived attribute specified using arithmetic expression $ArithExp$

   (a) if A is not versioned, then SQL_A is:
   `select` $A = ArithExp'$ `from` Rp_i `where` `_oid` $= oid$;
   where $ArithExp'$ is obtained by replacing every OPM attribute in $ArithExp$ by the relational attribute that represents it;

   (b) if A is versioned, and v is specified, then SQL_A is:
   `select` $A = ArithExp'$ `from` Rv_i `where` `_oid` $= oid$ `and` `_version` $= v$;
   where $ArithExp'$ is obtained by replacing every OPM attribute in $ArithExp$ by the relational attribute that represents it;

   (c) if A is versioned, and v is not specified (i.e., using default), then SQL_A is:
   `select` $A = ArithExp'$ `from` Rp_i, Rv_i
   `where` Rp_i`._oid` $= oid$ `and` Rv_i`._oid` $= oid$
   `and` Rp_i`._defaultVersion` $=$ Rv_i`._version`;
   where $ArithExp'$ is obtained by replacing every OPM attribute in $ArithExp$ by the relational attribute that represents it;

2. If A is an aggregate derived attribute specified using ρ applied on primitive numeric multi-valued attribute A' of O_i, where ρ is **count**, **sum**, **avg**, **min**, or **max**, and A' is represented by attribute B' of auxiliary relation $Ra_{A'}(Oid_i, B', \ldots)$

   (a) if A is not versioned, then SQL_A is:
   `select` $A = \rho\ (B')$ `from` $Ra_{A'}$ `where _oid` $=$ oid;

   (b) if A is not versioned, and v is specified, then SQL_A is:
   `select` $A = \rho\ (B')$ `from` $Ra_{A'}$ `where _oid` $=$ oid `and _version` $= v$;

   (c) if A is not versioned, and v is not specified (i.e., using default) then SQL_A is:
   `select` $A = \rho\ (B')$ `from` Rp_i, $Ra_{A'}$
   `where` Rp_i`._oid` $=$ oid `and` $Ra_{A'}$`._oid` $=$ oid
   `and` Rp_i`._defaultVersion` $=$ $Ra_{A'}$`._version`;

If A is inherited from superclass O_s, then the retrieval method for A is the same except that Rp_i is replaced by the primary relation for O_s, Rp_s, and Rv_i is replaced by the primary version relation for O_s, Rv_s.

4.4 Retrieval Method for Composition Derived Attribute

If A is a composition derived attribute of a derived or non-derived OPM class O_i, then a sequence of SQL subqueries is generated. Let composition path ($[O_0]$) C_0 $[O_1]$ C_1 $\ldots$ C_{m-1} $[O_m]$ be involved in the specification of A. (Class O_0 is included in the composition path only when O_i is a derived class.)

Retrieval method for A consists of an SQL query, SQL_A, constructed as follows: (If O_i is a derived class, then Rp_i below is the view representing O_i, and there is no Rv_i.)

First, start with C_0:

1. if C_0 is an attribute A_0

   (a) if A_0 is not versioned

      i. if A_0 is primitive and single-valued
      if A_0 is local to O_i (if O_i is non-derived) or O_0 (if O_i is derived), then subquery SQL_0 is:
      `select` $Rp_i.A_0$, `v#` $=$ `null from` Rp_i `where` Rp_i`._oid` $=$ oid;
      if A_0 is inherited from O_s, then replace Rp_i by Rp_s;

      ii. if A_0 is primitive and multi-valued, then subquery SQL_0 is:

         A. if O_i is non-derived:
         `select` $Ra_A.A_0$, `v#` $=$ `null from` Ra_A `where` Ra_A`.Oid` $=$ oid;

B. if O_i is derived:
`select` $Ra_A.A_0$, `v#` = `null` `from` Rp_i, Ra_A
`where` Rp_i.`_oid` = oid `and` Ra_A.`_oid` = oid;

iii. if A_0 is abstract and single-valued
if A is local to O_i (if O_i is non-derived) or O_0 (if O_i is derived), then SQL_0 is:
`select` $Rp_i.A_0$, `v#` = `null` `from` Rp_i `where` Rp_i.`_oid` = oid;
if A_0 is inherited from O_s, then replace Rp_i by Rp_s;

iv. if A_0 is abstract and multi-valued,

A. if O_i is non-derived:
`select` $Ra_A.A_0$, `v#` = `null` `from` Ra_A `where` Ra_A.`Oid` = oid;

B. if O_i is derived:
`select` $Ra_A.A_0$, `v#` = `null` `from` Rp_i, Ra_A
`where` Rp_i.`_oid` = oid `and` Ra_A.`_oid` = oid;

(b) if A_0 is versioned, and v is specified (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)

i. if A_0 is primitive, single-valued
if A_0 is local to O_i, then SQL_0 is:
`select` $Rv_i.A_0$, `v#` = `null` `from` Rv_i
`where` Rv_i.`_oid` = oid `and` Rv_i.`_version` = v;
if A_0 is inherited from O_s, then Rv_i is replaced by Rv_s in the above subquery;

ii. if A_0 is primitive and multi-valued, then SQL_0 is:
`select` $Ra_A.A_0$, `v#` = `null` `from` Ra_A
`where` Ra_A.`_oid` = oid `and` Ra_A.`_version` = v;

iii. if A_0 is abstract with generic reference, and single-valued
if A_0 is local to O_i, then SQL_0 is:
`select` $Rv_i.A_0$, Rp_1.`_defaultVersion` `from` Rv_i, Rp_1
`where` Rv_i.`_oid` = oid `and` Rv_i.`_version` = v
`and` $Rv_i.A_0$ = Rp_1.`_oid`;
if A_0 is inherited from O_s, then Rv_i is replaced by Rv_s in the above subquery;

iv. if A_0 is abstract with generic reference and multi-value, then SQL_0 is:
`select` $Ra_A.A_0$, Rp_1.`_defaultVersion` `from` Ra_A, Rp_1
`where` Ra_A.`_oid` = oid `and` Ra_A.`_version` = v
`and` $Ra_A.A_0$ = Rp_1.`_oid`;

v. if A_0 is abstract with specific reference, and single-valued
if A_0 is local to O_i, then SQL_0 is:
`select` $Rv_i.A_0$, $Rv_i.A_0$`_v#` `from` Rv_i `where` Rv_i.`_oid` = oid `and` Rv_i.`_version` = v;

if A_0 is inherited from O_s, then Rv_i is replaced by Rv_s in the above subquery;

vi. if A_0 is abstract with specific reference and multi-valued, then SQL_0 is:
`select` $Ra_A.A_0$, Rv_i.`_v`A_0 `from` Ra_A `where` Ra_A.`_oid` $= oid$ `and` Ra_A.`_version` $= v$;

(c) if A_0 is versioned, and v is not specified (i.e., using default) (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)

i. if A_0 is primitive, single-valued
if A_0 is local to O_i, then SQL_0 is:
`select` $Rv_i.A_0$, `v#` = `null` `from` Rp_i, Rv_i
`where` Rp_i.`_oid` $= oid$ `and` Rv_i.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` = Rv_i.`_version`;
if A_0 is inherited from O_s, then Rv_i is replaced by Rv_s in the above subquery;

ii. if A_0 is primitive and multi-valued, then SQL_0 is:
`select` $Ra_A.A_0$, `v#` = `null` `from` Rp_i, Ra_A
`where` Rp_i.`_oid` $= oid$ `and` Ra_A.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` = Ra_A.`_version`;

iii. if A_0 is abstract with generic reference, and single-valued
if A_0 is local to O_i, then SQL_0 is:
`select` $Rv_i.A_0$, Rp_1.`_defaultVersion` `from` Rp_i, Rv_i, Rp_1
`where` Rp_i.`_oid` $= oid$ `and` Rv_i.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` = Rv_i.`_version` `and` $Rv_i.A_0$ = Rp_1.`_oid`;
if A_0 is inherited from O_s, then Rv_i is replaced by Rv_s in the above subquery;

iv. if A_0 is abstract with generic reference and multi-value, then SQL_0 is:
`select` $Ra_A.A_0$, Rp_1.`_defaultVersion` `from` Rp_i, Ra_A, Rp_1
`where` Rp_i.`_oid` $= oid$ `and` Ra_A.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` = Ra_A.`_version` `and` $Ra_A.A_0$ = Rp_1.`_oid`;

v. if A_0 is abstract with specific reference, and single-valued
if A_0 is local to O_i, then SQL_0 is:
`select` $Rv_i.A_0$, Rv_i.`_v`A_0 `from` Rp_i, Rv_i
`where` Rp_i.`_oid` $= oid$ `and` Rv_i.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` = Rv_i.`_version`;
if A_0 is inherited from O_s, then Rv_i is replaced by Rv_s in the above subquery;

vi. if A_0 is abstract with specific reference and multi-valued, then SQL_0 is:
`select` $Ra_A.A_0$, Rv_i.`_v`A_0 `from` Rp_i, Ra_A
`where` Rp_i.`_oid` $= oid$ `and` Ra_A.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` = Ra_A.`_version`;

2. if C_0 is $!A_0$ (A_0 must be abstract)

(a) if A_0 is not versioned

i. if A_0 is single-valued

A. if both O_i and O_1 not versioned, then SQL_0 is:
`select` Rp_1.`_oid`, `v#` = `null from` Rp_1, Rp_i
`where` $Rp_1.A_0 = oid$ `and` Rp_i.`_oid` $= oid$;

B. if O_i is not versioned and O_1 is versioned
if A_0 is local to O_1, then SQL_0 is:
`select` Rp_1.`_oid`, Rp_1.`_defaultVersion from` Rp_1, Rp_i
`where` $Rp_1.A_0 = oid$ `and` Rp_i.`_oid` $= oid$;
if A_0 is inherited from superclass O_s of O_1, then SQL_0 is:
`select` Rp_1.`_oid`, Rp_1.`_defaultVersion from` Rp_1, Rp_s, Rp_i
`where` $Rp_s.A_0 = oid$ `and` Rp_1.`_oid` = Rp_s.`_oid`
`and` Rp_i.`_oid` $= oid$;

C. if O_i is versioned and O_1 is not versioned (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)
if v is not specified, then SQL_0 is:
`select` Rp_1.`_oid`, `v#` = `null from` Rp_1, Rp_i
`where` $Rp_1.A_0 = oid$ `and` Rp_i.`_oid` $= oid$;
if v is specified, then SQL_0 is:
`select` Rp_1.`_oid`, `v#` = `null from` Rp_1, Rp_i
`where` $Rp_1.A_0 = oid$ `and` Rp_i.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` $= v$;

D. if both O_i and O_1 are versioned, and A_0 is local to O_1 (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)
if v is not specified, then SQL_0 is:
`select` Rp_1.`_oid`, Rp_1.`_defaultVersion from` Rp_1, Rp_i
`where` $Rp_1.A_0 = oid$ `and` Rp_i.`_oid` $= oid$;
if v is specified, then SQL_0 is:
`select` Rp_1.`_oid`, Rp_1.`_defaultVersion from` Rp_1, Rp_i
`where` $Rp_1.A_0 = oid$ `and` Rp_i.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` $= v$;

E. if both O_i and O_1 are versioned, and A_0 is inherited from superclass O_s of O_1 (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)
if v is not specified, then SQL_0 is:
`select` Rp_1.`_oid`, Rp_1.`_defaultVersion from` Rp_1, Rp_s, Rp_i
`where` $Rp_s.A_0 = oid$ `and` Rp_1.`_oid` = Rp_s.`_oid`

and Rp_i.`_oid` = oid;
if v is specified, then SQL_0 is:
`select` Rp_1.`_oid`, Rp_1.`_defaultVersion from` Rp_1, Rp_i, Rp_s
`where` $Rp_s.A_0 = oid$ `and` Rp_s.`_oid` = Rp_1.`_oid`
Rp_i.`_oid` = oid `and` Rp_i.`_defaultVersion` = v;

ii. if A_0 is multi-valued

A. if both O_i and O_1 not versioned, then SQL_0 is:
`select` Ra_A.`_oid`, `v#` = `null from` Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Rp_i.`_oid` = oid;

B. if O_i is not versioned and O_1 is versioned
`select` Rp_1.`_oid`, Rp_1.`_defaultVersion from` Rp_1, Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Rp_1.`_oid` = Ra_A.`_oid`
`and` Rp_i.`_oid` = oid;

C. if O_i is versioned and O_1 is not versioned (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)
if v is not specified, then SQL_0 is:
`select` Ra_A.`_oid`, `v#` = `null from` Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Rp_i.`_oid` = oid;
if v is specified, then SQL_0 is:
`select` Ra_A.`_oid`, `v#` = `null from` Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Rp_i.`_oid` = oid
`and` Rp_i.`_defaultVersion` = v;

D. if both O_i and O_1 are versioned (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)
if v is not specified, then SQL_0 is:
`select` Ra_A.`_oid`, Rp_1.`_defaultVersion from` Rp_1, Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Rp_1.`_oid` = Ra_A.`_oid`
`and` Rp_i.`_oid` = oid;
if v is specified, then SQL_0 is:
`select` Rp_1.`_oid`, Rp_1.`_defaultVersion from` Rp_1, Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Rp_1.`_oid` = Ra_A.`_oid`
`and` Rp_i.`_oid` = oid `and` Rp_i.`_defaultVersion` = v;

(b) if A_0 is versioned and with generic reference (class O_1 must be versioned)

i. if A_0 is single-valued

A. if O_i is not versioned, then SQL_0 is:
`select` Rv_1.`_oid`, Rv_1.`_version from` Rv_1, Rp_i
`where` $Rv_1.A_0 = oid$ `and` Rp_i.`_oid` = oid;

if A_0 is inherited from superclass O_s of O_1, then Rv_1 in above subquery is replaced by Rv_s;

B. if O_i is versioned, and v is not specified (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)
`select` Rv_1.`_oid`, Rv_1.`_version from` Rv_1. Rp_i
`where` $Rv_1.A_0 = oid$ `and` Rp_i.`_oid` $= oid$;
if A_0 is inherited from superclass O_s of O_1, then Rv_1 in above subquery is replaced by Rv_s;

C. if O_i is versioned, and v is specified (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)
`select` Rv_1.`_oid`, Rv_1.`_version from` Rv_1, Rp_i
`where` $Rv_1.A_0 = oid$ `and` Rp_i.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` $= v$;
if A_0 is inherited from superclass O_s of O_1, then Rv_1 in above subquery is replaced by Rv_s;

ii. if A_0 is multi-valued

A. if O_i is not versioned, then SQL_0 is:
`select` Ra_A.`_oid`, Ra_A.`_version from` Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Rp_i.`_oid` $= oid$;

B. if O_i is versioned, and v is not specified (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)
`select` Ra_A.`_oid`, Ra_A.`_version from` Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Rp_i.`_oid` $= oid$;

C. if O_i is versioned, and v is specified (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)
`select` Ra_A.`_oid`, Ra_A.`_version from` Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Rp_i.`_oid` $= oid$
`and` Rp_i.`_defaultVersion` $= v$;

(c) if A_0 is versioned and with specific reference (both class O_1 and class O_i must be versioned) (OPM 4.0 only allows non-versioned derived classes, and therefore in this case, O_i must be non-derived.)

i. if A_0 is single-valued

A. if v is not specified
`select` Rv_1.`_oid`, Rv_1.`_version from` Rv_1, Rp_i
`where` Rp_i.`_oid` $= oid$ `and` $Rv_1.A_0 = oid$
`and` Rv_1.`_v`$A_0 = Rp_i$.`_defaultVersion`;

if A_0 is inherited from superclass O_s of O_1, then Rv_1 in above subquery is replaced by Rv_s;

B. if v is specified
`select` Rv_1`.Oid`, Rv_1`._version` `from` Rv_1, Rp_i
`where` $Rv_1.A_0 = oid$ `and` Rv_1`._v`$A_0 = v$
`and` Rp_i`._oid` $= oid$;
if A_0 is inherited from superclass O_s of O_1, then Rv_1 in above subquery is replaced by Rv_s;

ii. if A_0 is multi-valued

A. if v is not specified
`select` Ra_A`._oid`, Ra_A`._version` `from` Ra_A, Rp_i
`where` Rp_i`._oid` $= oid$ `and` $Ra_A.A_0 = oid$
`and` Ra_A`._v`$A_0 = Rp_i$`._defaultVersion`;

B. if v is specified
`select` Ra_A`._oid`, Ra_A`._version` `from` Ra_A, Rp_i
`where` $Ra_A.A_0 = oid$ `and` Ra_A`._v`$A_0 = v$
`and` Rp_i`._oid` $= oid$;

Suppose SQL_{k-1} generated until C_{k-1} in composition path is:

`select` Oid_{k-1}, Ver_{k-1} `from` RS_{k-1} `where` $Cond_{k-1}$;

where RS_{k-1} is a list of relation names, and $Cond_{k-1}$ is an and-ed condition. SQL_k for composition path until C_k is constructed as follows:

1. if C_k is an attribute A_k

(a) if A_k is not versioned

i. if A_k is primitive and single-valued
if A_k is local to O_k, then subquery SQL_k is:
`select` $Rp_k.A_k$, `v#` $=$ `null` `from` Rp_k, RS_{k-1}
`where` Rp_k`._oid` $= Oid_{k-1}$ `and` $Cond_{k-1}$;
if A_k is inherited from O_s, then replace Rp_k by Rp_s;

ii. if A_k is primitive and multi-valued, then subquery SQL_k is:
`select` $Ra_A.A_k$, `v#` $=$ `null` `from` Ra_A, RS_{k-1}
`where` Ra_A`._oid` $= Oid_{k-1}$ `and` $Cond_{k-1}$;

iii. if A_k is abstract and single-valued
if A_k is local to O_k, then SQL_k is:
`select` $Rp_k.A_k$, `v#` $=$ `null` `from` Rp_k, RS_{k-1}
`where` Rp_k`._oid` $= Oid_{k-1}$ `and` $Cond_{k-1}$;
if A_k is inherited from O_s, then replace Rp_k by Rp_s;

iv. if A_k is abstract and multi-valued,
`select` $Ra_A.A_k$, `v#` $=$ `null from` Ra_A, RS_{k-1}
`where` Ra_A.`_oid` $= Oid_{k-1}$ `and` $Cond_{k-1}$;

(b) if A_k is versioned

i. if A_k is primitive, single-valued
if A_k is local to O_k, then SQL_k is:
`select` $Rv_k.A_k$, `v#` $=$ `null from` Rv_k, RS_{k-1}
`where` Rv_k.`_oid` $= Oid_{k-1}$ `and` Rv_k.`_version` $= Ver_{k-1}$
`and` $Cond_{k-1}$;
if A_k is inherited from O_s, then Rv_k is replaced by Rv_s in the above subquery;

ii. if A_k is primitive and multi-valued, then SQL_k is:
`select` $Ra_A.A_k$, `v#` $=$ `null from` Ra_A, RS_{k-1}
`where` Ra_A.`_oid` $= Oid_{k-1}$ `and` Ra_A.`_version` $= Ver_{k-1}$
`and` $Cond_{k-1}$;

iii. if A_k is abstract with generic reference, and single-valued
if A_k is local to O_k, then SQL_k is:
`select` $Rv_k.A_k$, Rp_{k+1}.`_defaultVersion from` Rv_k, Rp_1, RS_{k-1}
`where` Rv_k.`_oid` $= Oid_{k-1}$ `and` Rv_k.`_version` $= Ver_{k-1}$
`and` $Rv_i.A_k = Rp_{k+1}$.`_oid and` $Cond_{k-1}$;
if A_k is inherited from O_s, then Rv_k is replaced by Rv_s in the above subquery;

iv. if A_k is abstract with generic reference and multi-value, then SQL_k is:
`select` $Ra_A.A_k$, Rp_{k+1}.`_defaultVersion from` Ra_A, Rp_1, RS_{k-1}
`where` Ra_A.`_oid` $= Oid_{k-1}$ `and` Ra_A.`_version` $= Ver_{k-1}$
`and` $Ra_A.A_k = Rp_{k+1}$.`_oid and` $Cond_{k-1}$;

v. if A_k is abstract with specific reference, and single-valued
if A_k is local to O_k, then SQL_k is:
`select` $Rv_k.A_k$, Rv_k.`_v`A_k `from` Rv_k, RS_{k-1}
`where` Rv_k.`_oid` $= Oid_{k-1}$ `and` Rv_k.`_version` $= Ver_{k-1}$
`and` $Cond_{k-1}$;
if A_k is inherited from O_s, then Rv_k is replaced by Rv_s in the above subquery;

vi. if A_k is abstract with specific reference and multi-valued, then SQL_k is:
`select` $Ra_A.A_k$, Rv_k.`_v`A_k `from` Ra_A, RS_{k-1}
`where` Ra_A.`_oid` $= Oid_{k-1}$ `and` Ra_A.`_version` $= Ver_{k-1}$
`and` $Cond_{k-1}$;

2. if C_k is $!A_k$ (A_k must be abstract)

(a) if A_k is not versioned

i. if A_k is single-valued

A. if both O_k and O_{k+1} not versioned, then SQL_k is:
`select` Rp_{k+1}`._oid`, `v# = null from` Rp_{k+1}, RS_{k-1}
`where` $Rp_{k+1}.A_k = Oid_{k-1}$ `and` $Cond_{k-1}$;

B. if O_k is not versioned and O_{k+1} is versioned
if A_k is local to O_{k+1}, then SQL_k is:
`select` Rp_{k+1}`._oid`, Rp_{k+1}`._defaultVersion from` Rp_{k+1}, RS_{k-1}
`where` $Rp_{k+1}.A_k = Oid_{k-1}$ `and` $Cond_{k-1}$;
if A_k is inherited from superclass O_s of O_k, then SQL_k is:
`select` Rp_{k+1}`._oid`, Rp_{k+1}`._defaultVersion from` Rp_{k+1}, Rp_s, RS_{k-1}
`where` $Rp_s.A_k = Oid_{k-1}$ `and` Rp_s`._oid =` Rp_{k+1}`._oid`
`and` $Cond_{k-1}$;

C. if O_k is versioned and O_{k+1} is not versioned, then SQL_k is:
`select` Rp_{k+1}`._oid`, `v# = null from` Rp_k, Rp_{k+1}, RS_{k-1}
`where` $Rp_{k+1}.A_k = Oid_{k-1}$ `and` Rp_k`._oid =` Oid_{k-1}
`and` Rp_k`._defaultVersion =` Ver_{k-1} `and` $Cond_{k-1}$;

D. if both O_k and O_{k+1} are versioned
if A_k is local to O_{k+1}, then SQL_k is:
`select` Rp_{k+1}`._oid`, Rp_{k+1}`._defaultVersion from` Rp_k, Rp_{k+1}, RS_{k-1}
`where` $Rp_{k+1}.A_k = Oid_{k-1}$ `and` Rp_k`._oid =` Oid_{k-1}
`and` Rp_k`._defaultVersion =` Ver_{k-1} `and` $Cond_{k-1}$;
if A_k is inherited from superclass O_s of O_{k+1}, then SQL_k is:
`select` Rp_{k+1}`._oid`, Rp_{k+1}`._defaultVersion from` Rp_k, Rp_{k+1}, Rp_s, RS_{k-1}
`where` $Rp_s.A_k = Oid_{k-1}$ `and` Rp_{k+1}`._oid =` Rp_s`._oid`
`and` Rp_k`._oid =` Oid_{k-1} `and` Rp_k`._defaultVersion =` Ver_{k-1}
`and` $Cond_{k-1}$;

ii. if A_k is multi-valued

A. if both O_k and O_{k+1} not versioned, then SQL_k is:
`select` Ra_A`._oid`, `v# = null from` Ra_A, RS_{k-1}
`where` $Ra_A.A_k = oid$ `and` $Cond_{k-1}$;

B. if O_k is not versioned and O_{k+1} is versioned
`select` Rp_{k+1}`._oid`, Rp_{k+1}`._defaultVersion from` Rp_{k+1}, Ra_A, RS_{k-1}
`where` $Ra_A.A_k = Oid_{k-1}$ `and` Rp_{k+1}`._oid =` Ra_A`._oid`
`and` $Cond_{k-1}$;

C. if O_k is versioned and O_{k+1} is not versioned, then SQL_k is:
`select` Ra_A`._oid`, `v# = null from` Ra_A, Rp_k, RS_{k-1}
`where` $Ra_A.A_k = Oid_{k-1}$ `and` Rp_k`._oid =` Oid_{k-1}
`and` Rp_k`._defaultVersion =` Ver_{k-1} `and` $Cond_{k-1}$;

D. if both O_k and O_{k+1} are versioned, then SQL_k is:
`select` Rp_{k+1}`._oid`, Rp_{k+1}`._defaultVersion from` Rp_{k+1}, Ra_A, Rp_k, RS_{k-1}
`where` $Ra_A.A_k = Oid_{k-1}$ `and` Rp_{k+1}`._oid` $=$ Ra_A`._oid`
`and` Rp_k`._oid` $= Oid_{k-1}$ `and` Rp_k`._defaultVersion` $= Ver_{k-1}$
`and` $Cond_{k-1}$;

(b) if A_k is versioned and with generic reference (class O_{k+1} must be versioned)

i. if A_k is single-valued

A. if O_k is not versioned
if A_k is local to O_{k+1}, then SQL_k is:
`select` Rv_{k+1}`._oid`, Rv_{k+1}`._version from` Rv_{k+1}, RS_{k-1}
`where` $Rv_{k+1}.A_k = Oid_{k-1}$ `and` $Cond_{k-1}$;
if A_k is inherited from superclass O_s of O_{k+1}, then Rv_{k+1} above is replaced by Rv_s;

B. if O_k is versioned
`select` Rv_{k+1}`._oid`, Rv_{k+1}`._version from` Rv_k, Rp_{k+1}, RS_{k-1}
`where` $Rv_{k+1}.A_k = Oid_{k-1}$ `and` Rp_k`._oid` $= Oid_{k-1}$
`and` Rp_k`._defaultVersion` $= Ver_{k-1}$ `and` $Cond_{k-1}$;
if A_k is inherited from superclass O_s of O_{k+1}, then Rv_{k+1} above is replaced by Rv_s;

ii. if A_k is multi-valued

A. if O_k is not versioned, then SQL_k is:
`select` Ra_A`._oid`, Ra_A`._version from` Ra_A, RS_{k-1}
`where` $Ra_A.A_k = oid$ `and` $Cond_{k-1}$;

B. if O_k is versioned
`select` Ra_A`._oid`, Ra_A`._version from` Ra_A, Rp_k, RS_{k-1}
`where` $Ra_A.A_k = Oid_{k-1}$ `and` Rp_k`._oid` $= Oid_{k-1}$
`and` Rp_k`._defaultVersion` $= Ver_{k-1}$ `and` $Cond_{k-1}$;

(c) if A_k is versioned and with specific reference (both class O_k and class O_{k+1} must be versioned)

i. if A_k is single-valued
`select` Rv_{k+1}`._oid`, Rv_{k+1}`._version from` Rv_{k+1}, RS_{k-1}
`where` $Rv_{k+1}.A_k = Oid_{k-1}$ `and` $Rv_{k+1}.A_k$`._v#` $= Ver_{k-1}$
`and` $Cond_{k-1}$;
if A_k is inherited from superclass O_s of O_{k+1}, then Rv_{k+1} above is replaced by Rv_s;

ii. if A_k is multi-valued
`select` Ra_A`._oid`, Ra_A`._version from` Ra_A, RS_{k-1}
`where` $Ra_A.A_k = Oid_{k-1}$ `and` Ra_A`._v`$A_k = Ver_{k-1}$
`and` $Cond_{k-1}$;

Additional work has to be done on SQL_{m-1} to get the final SQL query for A, SQL_A. Suppose SQL_{m-1} is:
`select` Oid_{m-1}, Ver_{m-1} `from` RS_{m-1} `where` $Cond_{m-1}$;

1. if C_{m-1} is attribute A_{m-1}
   (a) if O_m is the only value class for A_{m-1}
      i. if O_m does not have REP attributes, then SQL_A is SQL_{m-1}
      ii. if O_m has REP attributes $D_1, \ldots, D_d$, then SQL_A is:
      `select` $D_1, \ldots, D_d$, Ver_{m-1} `from` RS_{m-1}, Rp_m `where` Rp_m.`_oid` $= Oid_{m-1}$ `and` $Cond_{m-1}$;
   (b) if O_m is subset/subclass of value class of A_{m-1}
      i. if O_m does not have REP attributes, then SQL_A is:
      `select` Rp_m.`_oid`, Ver_{m-1} `from` Rp_m, RS_{m-1} `where` Rp_m.`_oid` $= Oid_{m-1}$ `and` $Cond_{m-1}$;
      ii. if O_m has REP attributes $D_1, \ldots, D_d$, then SQL_A is:
      `select` $D_1, \ldots, D_m$, Ver_{m-1} `from` Rp_m, RS_{m-1} `where` Rp_m.`_oid` $= Oid_{m-1}$ `and` $Cond_{m-1}$;
2. if C_{m-1} is $!A_{m-1}$
   (a) if O_m does not have REP attributes, then SQL_A is SQL_{m-1}
   (b) if O_m has REP attributes $D_1, \ldots, D_d$, then SQL_A is:
   `select` $D_1, \ldots, D_d$, Ver_{m-1} `from` RS_{m-1}, Rp_m `where` Rp_m.`_oid` $= Oid_{m-1}$ `and` $Cond_{m-1}$;

Example. Retrieval method for derived attribute `CourseBooks` of class `FACULTY`, `get_CourseBooks(@oid)` is: is constructed as follows:
SQL_0 is:

```
select _vCOURSE._oid, _vCOURSE._version
from _vCOURSE
where _vCOURSE.Taughtby = @oid;
```

SQL_1 is:

```
select COURSE_References.References, BOOK._defaultVersion
from _vCOURSE, COURSE_References, BOOK
where _vCOURSE.Taughtby = @oid
and COURSE_References._oid = _vCOURSE._oid
and COURSE_References._version = _vCOURSE._version
and COURSE_References.References = BOOK._oid;
```

The final result (SQL_2) is:

```
select BOOK.Title, v# = null
from _vCOURSE, COURSE_References, BOOK
where _vCOURSE.Taughtby = @oid
and COURSE_References._oid = _vCOURSE._oid
and COURSE_References._version = _vCOURSE._version
and COURSE_References.References = BOOK._oid;
```

5 Data Manipulation Methods

In this section, we examine the problems involved in representing update methods in a relational database. There are six update methods:

1. insert new instance or new version;
2. physical delete an instance or a version;
3. local delete a version;
4. logical undelete a version;
5. update an instance or a version;
6. set default version.

Note that these update methods are only applicable to non-derived classes. Derived subclasses and derived superclasses cannot be updated.

5.1 Insert Method

Method insert([*oid*], val_1, ..., val_n) adds a new instance x or a new version y to class O_i, where *oid* is the object identity and val_1, ..., val_n are values (or Oid's) for attributes of O_i. This method returns the Oid value (and version number) of the new instance/version.

If x is a new instance to the database, then *oid* is automatically generated, and is not specified in the method.

```
insert OBJECTS values ();
select @_oid = @@identity;
```

If x is an existing instance of superclass of O_i, then *oid* must be specified. There must be an existing instances in the database with the same Oid, and therefore

```
select @_oid = _oid from OBJECTS
where _oid = _oid;
```

cannot return null.

Assume O_i is a base class.

1. if x is an new instance to the database
   (a) if O_i is not versioned
   First, insert all single-valued attributes of O_i.
   ```
   insert Rp_i (_oid, A_1, ..., A_k)
   values (@_oid, val_1, ..., val_k);
   ```
   Then insert (the first) values to multi-valued attributes A_m:
   ```
   insert Ra_m (_oid, A_m)
   ```

`values` (`@_oid`, val_m);

If A_m is list-valued, then

`insert` Ra_m (`_oid`, `_order`, A_m)

`values` (`@_oid`, 1, val_m);

(b) if O_i is versioned

Insert values for all the non-versioned attributes as described above. The `_defaultVersion` in Rp_i is set to 1. Then insert the first version as follows:

First, insert all single-valued versioned attributes of O_i.

`insert` Rv_i (`_oid`, `_version`, `_fromDate`, Av_1, ..., Av'_k)

`values` (`@_oid`, 1, `@_t_now`, val'_1, ..., val'_k);

where `@_t_now` is the current data/time provided by the user.

Then insert (the first) values to multi-valued attributes Av_m:

`insert` Ra_m (`_oid`, `_version`, A_m)

`values` (`@_oid`, 1, val'_m);

If Av_m is list-valued, then

`insert` Ra_m (`_oid`, `_version`, `_order`, A_m)

`values` (`@_oid`, 1, 1, val'_m);

2. if x already exists in database

(a) if O_i is not versioned, then there is an error because O_i is a base class;

(b) if O_i is versioned, then only versioned attribute values can be inserted.

First, get the new version number as follows:

`select @_version =` `_nextVersion from` Rp_i

`where _oid = @_oid;`

And get the previous version number as follows:

`select @_preversion =` `_defaultVersion from` Rp_i

`where _oid = @_oid;`

First, insert all single-valued versioned attributes of O_i.

`insert` Rv_i (`_oid`, `_version`, `_preVersion`, `_fromDate`, Av_1, ..., Av'_k)

`values` (`@_oid`, `@_version`, `@_preversion`, `@_t_now`, val'_1, ..., val'_k);

Then insert (the first) values to multi-valued attributes Av_m:

`insert` Ra_m (`_oid`, `_version`, A_m)

`values` (`@_oid`, `@_version`, val'_m);

If Av_m is list-valued, then

`insert` Ra_m (`_oid`, `_version`, `_order`, A_m)

`values` (`@_oid`, `@_version`, 1, val'_m);

If O_i is a subclass of O_s, then inherited attributes from O_s must be inserted into associated relational tables for O_s first as described above.

All attributes of O_i specified as `not null` must appear in the insert statement associated with non-null values. If O_i is a subclass, then all inherited attributes of O_i that are specified as `not null` must also appear in the insert statement associated with non-null values.

If attribute A of O_i has value class O_j and is specified as `inverse of` attribute A' of O_j, where A' is (physically) represented by relational attribute B' then the A value is not inserted into a relation representing O_i, but rather into a relation representing O_j. Let y be an abstract value for A for instance x, let id be the `_oid` value for x and let id' be the `_oid` value for y.

Suppose A' is represented by B' in relation R_j(`_oid`, B', ...).

1. If A' is single-valued and B' value for `_oid` $= id'$ in R_j is
   (a) null, then the following SQL statement is generated:
   `update` R_j `set` $B' = id$
   `where _oid` $= id'$
   (b) equal to id, then no additional insertion is needed;
   (c) is not null nor is equal to id then this is an error;
2. if A' is multi-valued and id is not already a B' value for `_oid` $= id'$ in R_j then the following SQL statement is generated:
   `insert` R_j (`_oid`, B')
   `values` (id', id)

5.2 Deletion Methods

Method `delete`(oid) removes instance x of class O_i with identity (and all its versions) from database.

Deleting an instance x (with `_oid` $= id$) from class O_i involves deleting all tuples in the relations R_i representing O_i:

`delete` R_i `where _oid` $= oid$;

Deleting an instance of class O_i is propagated to all subclasses of O_i, but is not propagated to superclasses of O_i.

Delete propagation to auxiliary relations representing O_i can also be carried out by the database procedures (e.g., Sybase triggers) implementing delete cascade rules associated with the foreign keys in these auxiliary relations. Similarly, delete propagation to subclasses of O_i can be carried out by the database procedures implementing delete cascade rules associated with the foreign keys in the primary relations associated with the subclasses.

Method delete_version(oid, [v]) removes version v of instance x with identity oid in a class O_i. For all the primary/auxiliary version table R_i for O_i:

delete R_i where _oid = oid
and _version = v;

If v is not specified, then the default version is deleted:

select @_current = _defaultVersion from Rp_i
where _oid = oid;
delete R_i
where _oid = oid and _version = @_current;
update Rp_i set _defaultVersion = NULL
where _oid = oid;

5.3 Logical Delete Method

Method ldelete(oid, [v]) logically delete version v of instance x with identity oid in class O_i by setting the delete date to current date/time.

The current date/time is returned by:

select @_t_now = getdate();

The logical deletion in O_i is performed by:

update Rv_i set _deleteDate = @_t_now
where _oid = oid and _version = v;

If the version number v is not specified, then the current version is deleted:

update Rv_i set _deleteDate = @_t_now
from Rp_i, Rv_i
where Rp_i._oid = oid and Rv_i._oid = oid
and Rp_i._defaultVersion = Rv_i._version;

The current version information in Rp_i is also removed:

update Rp_i set _defaultVersion = NULL
where _oid = oid and _defaultVersion = v;

The logical deletion is propagated to all subclasses of O_i.

5.4 Logical Undelete Method

Method l_undelete(oid, v) undeletes version v of instance x with identity oid in class O_i. It is implemented by removing delete date:

update Rv_i set _deleteDate = NULL
where _oid = oid and _version = v;

The logical undelete propagated to all superclasses of O_i as well.

5.5 Attribute Value Update Methods

Update of a single-valued local attribute, A, of a class O_i is carried out by an SQL update statement for setting the new value of the attribute representing A in Rp_i or in Rv_i.

Updating a multi-valued local attribute of O_i, A, is performed by inserting and/or deleting tuples in the auxiliary relation representing A.

The update stored procedure only replaces values in an instance or in a version. It does not create new versions.[6] In order to create a new version for changes, the insert method should be used to create a new version, and then replace values in this new version.

5.6 Set Default Version Method

Method set_current(oid, v) in class O_i set version v to be the current version for instance x with identity oid.

First check if v is an existing non-deleted version for instance x:

```
select _oid, _version from Rv_i
where _oid = oid and _version = v
and _deleteDate is null;
```

If the above select query returns null, then give an error message; otherwise set new default current as follows:

```
update Rp_i set _defaultVersion = v
where _oid = oid;
```

[6]Since updating a version may be done through several procedure calls, we do not want to create new version for each procedure call.

6 The Effect of Permission Modeling on Methods

If an OPM schema supports permission modeling, then additional conditions must be included into the methods described in the previous two sections.

Suppose that *owner* and *status* attributes used for modeling permissions are named `owner` and `status`, respectively. each retrieval method described in section 4, the following conditions must be added to each table that contains the (de-normalized) attributes representing `owner` and `permission` OPM attributes:

(`owner` = `@owner` OR `status` > 0)

Only the owner of an object x can perform the six manipulation operations described in section 5 on this object x. Non-owners are not allowed to perform updates.

Four more retrieval and update operations are applicable to an object class O_i with permission mode *on*: (1) get owner, (2) update owner, (3) get status, and (4) update status. Only *super users* (determined by the DBA) are allowed to perform the first two operations, and only the owner of an object x is allowed to perform the last two operations on the object x.

The four operations for an object class O_i with permission mode *on*, are described below. Suppose Rp_i is the primary relation representing class O_i. Attributes `owner` and `status` are represented in relation Rp_i as well as by de-normalized attributes in auxiliary and version relations representing O_i.

6.1 Get Owner

Method get_owner(*oid*) returns the owner of an object x in class O_i with object identity equal to *oid*.

select owner from Rp_i
where _oid = *oid*;

6.2 Update Owner

Method update_owner(*oid*, *new_owner*) updates the owner of *oid* to a new owner *new_owner*.

Since an object x is not allowed to reference an un-released object y that does not belong to the same owner who owns x, checking must be performed to make sure no such references exist.

Then, the `owner` attribute in Rp_i is updated as follows:

update Rp_i set owner = *new_owner*
where _oid = *oid*;

All auxiliary and version tables Rx_i representing O_i must be updated as well:

update Rx_i set owner = *new_owner*
where _oid = *oid*;

The updates should also be propagated to all relations representing superclasses and subclasses of class O_i.

6.3 Get Status

Method get_status(oid, $owner$) returns the status of an object with identity oid. Only owners can perform this operation.

```
select status from Rp_i
where owner = owner;
```

6.4 Update Status

Method update_status(oid, $owner$, new_status) updates the status of an object with identity oid to new status new_status.

Since an object x is not allowed to reference an un-released object y in class O_i that does not belong to the same owner who owns x, checking must be performed to make sure no such y exists.

Then, the status attribute in Rp_i is updated as follows:

```
update Rp_i set status = new_status
where _oid = oid and owner = owner;
```

All auxiliary and version tables Rx_i representing O_i must be updated as well:

```
update Rx_i set status = new_status
where _oid = oid and owner = owner;
```

The updates should also be propagated to all relations representing superclasses and subclasses of class O_i.

7 Conclusion

We have described in this document the mapping of OPM schemas (OPM version 4.1) into abstract (DBMS-independent) relational schemas and SQL queries.

Based on the mapping procedure presented in this document, a tool for automatically mapping OPM schemas into abstract (DBMS-independent) relational schemas and SQL queries, and then into relational DBMS-specific schemas and queries, has been implemented using C++; this tool is described in [Chen & Markowitz 96b].

OPM tools, including the schema translator, are available via World Wide Web at `http://gizmo.lbl.gov/opm.html`.

References

[Chen & Markowitz 96a] Chen, I.A., and Markowitz, V.M., The Object-Protocol Model (Version 4.1), Lawrence Berkeley Laboratory Technical Report LBL-32738, 1996.

[Chen & Markowitz 96b] Chen, I.A., and Markowitz, V.M., OPM Schema Translator 4.1, Reference Manual, Lawrence Berkeley Laboratory Technical Report LBL-35582, 1996.

[Date 86] Date, C.J., Referential Integrity, in *Relational Database-Selected Writings*, Addison-Wesley, 1986.

[Galindo-Jegaria & Rosenthal 92] Galindo-Jegaria, C., and Rosenthal, A., How to Extend a Conventional Optimizer to Handle One- and Two-Sided Outerjoin, *Proc. of the 8th Int. Conf. on Data Engineering*, pp. 402-409, 1992.

[Hull & King 87] Hull, R., and King, R., Semantic Database Modeling: Survey, Applications, and Research Issues", *Computing Surveys* **19**, 3 (Sep. 1987), pp. 201-260.

[Maier 83] Maier, D., *The Theory of Relational Databases*, Computer Science Press, 1983.

[ODMG-93] *The Object Database Standard: ODMG-93*, Release 1.2. Cattell, R. G. G. (ed), Morgan Kaufmann, 1996.

A The Syntactic Definition for OPM Schemas

A.1 BNF Grammar for OPM

```
<OPM schema> ::= <OPM schema description>
                 <foreign database information>
                 <OPM classes>
               ;
<OPM schema description> ::= <null>
                           | SCHEMA <schema name>
                                    <schema version>
                                    <database name>
                                    <schema description>
                                    <permissions>
                           ;
<schema name> ::= <null>
                | <string>
                ;
<schema version> ::= <null>
                   | VERSION ':' <string>
                   ;
<database name> ::= <null>
                  | DATABASE ':' <string>
                  ;
<schema description> ::= <null>
                       | DESCRIPTION ':' <text>
                       ;
<permissions> ::= <null>
                | PERMISSIONS <owner attribute> <status attribute>
                ;
<owner attribute> ::= OWNER <simple attribute name> ':'
                      <abstract class list>
                      <attribute description>
                      <attribute example>
                      <domain-specific properties>
                    ;
<status attribute> ::= STATUS <simple attribute name> ':'
                      <status value class>
                      <simple default>
                      <attribute description>
                      <attribute example>
                      <domain-specific properties>
```

```
                    ;
<status value class> ::= <controlled value class name>
                       | <primitive value class>
                       | <named primitive value class>
                       ;
<foreign database information> ::= <foreign databases>
                                 | <null>
                                 ;
<foreign databases> ::= FOREIGN_DATABASES: { }
                      | FOREIGN_DATABASES: { <foreign dbs> }
                      ;
<foreign dbs> ::= ( <foreign database name> , <foreign description> )
                | <foreign dbs> , ( <foreign database name> , <foreign description> )
                ;
<foreign description> ::= <text>
                        | <null>
                        ;
<foreign database name> ::= <text>
                          ;
<OPM classes> ::= <OPM class>
                | <OPM classes> <OPM class>
                ;
<OPM class> ::= <controlled value class>
              | <named primitive value class>
              | <object class> <versioned attributes>
              | <object class> <object attributes> <versioned attributes>
              | <protocol class> <protocol attributes>
                <versioned protocol attributes>
              | <OPM derived class>
              ;
<object class> ::= <is foreign> OBJECT CLASS <object class name>
                   <isa list or null>
                   <class description>
                   <class example>
                   <domain-specific properties>
                   <object identifier>
                   <represented by>
                   <object class delete rules>
                   <foreign info>
                 ;
<is foreign> ::= FOREIGN
               | <null>
```

```
              ;
<isa list or null> ::= <generic object classes>
                     | <null>
                     ;
<object identifier> ::= ID : <attribute list>
                      | ID : ( <comp attribute list > )
                      | <null>
                      ;
<controlled value class> ::= <controlled value class name> { <enum-constant> }
                             <class default> <code type>
                             <class description>
                             <class example>
                             <domain-specific properties>
                           | <controlled value class name> { <ranges> }
                             <class default> <code type>
                             <class description>
                             <class example>
                             <domain-specific properties>
                           ;
<controlled value class name> ::= CONTROLLED VALUE CLASS <class name>
                                ;
<enum-constant> ::= <constant_code_desc>
                  | <enum-constant> , <constant_code_desc>
                  ;
<constant_code_desc> ::= <constant>
                       | ( <constant> , <code> )
                       | ( <constant> , <value description> )
                       | ( <constant> , <code> , <value description> )
                       ;
<value description> ::= <text>
                      ;
<ranges> ::= <range_desc>
           | <ranges> , <range_desc>
           ;
<range_desc> ::= <range>
               | ( <range> , <value description> )
               ;
<range> ::= <number>
          | ( - <number> )
          | <number> - <number>
          | ( - <number> ) - <number>
          | ( - <number> ) - ( - <number> )
```

```
          ;
<code type> ::= <null>
              | CODE_TYPE ':' <code data type>
              ;
<code data type> ::= CHAR '(' <integer> ')'
                   | VARCHAR '(' <integer> ')'
                   | INTEGER
                   | SMALLINT
                   | TINYINT
                   | REAL
                   | FLOAT | FLOAT ( <integer> )
                   | DECIMAL | DECIMAL '(' <integer> ')'
                   | DECIMAL '(' ( <integer> ',' <integer> ')'
                   | NUMERIC | NUMERIC '(' <integer> ')'
                   | NUMERIC '(' ( <integer> ',' <integer> ')'
                   ;
<named primitive value class> ::= NAMED PRIMITIVE VALUE CLASS <class name> :
                                  <primitive value class>
                                  <class default>
                                  <class description>
                                  <domain-specific properties>
                                ;
<class default> ::= DEFAULT : <default value>
                  | <null>
                  ;
<object attributes> ::= <object attribute>
                      | <object attributes> <object attribute>
                      ;
<protocol attributes> ::= <protocol attribute>
                        | <protocol attributes> <protocol attribute>
                        ;
<object attribute> ::= <simple attribute>
                     | <tuple attribute>
                     | <derived attribute>
                     ;
<protocol attribute> ::= <simple attribute>
                       | <tuple attribute>
                       | <derived attribute>
                       | <input or output attribute>
                       ;
<simple attribute> ::= ATTRIBUTE <simple attribute name> :
                                 <single or set or list>
```

```
                                <cardinality> <value class>
                                <simple default>
                                <inverse constraint>
                                <simple order by>
                                <attribute delete rule>
                                <attribute description>
                                <attribute example>
                                <domain-specific properties>
                    ;
<simple attribute name> ::= <attribute name>
                         ;
<tuple attribute> ::= ATTRIBUTE <tuple attribute name>
                                   ( <comp attribute list> ) :
                                   <single or set or list> <cardinality>
                                   ( <comp value classes> )
                                   <tuple default>
                                   <comp null rule>
                                   <component inverse constraints>
                                   <tuple order by>
                                   <attribute delete rule>
                                   <attribute description>
                                   <domain-specific properties>
                       ;
<tuple attribute name> ::= <attribute name>
                        | <null>
                        ;
<comp attribute list> ::= <comp attribute name>
                       | <comp attribute list> , <comp attribute name>
                       ;
<comp attribute name> ::= <attribute name>
                       ;
<comp value classes> ::= <value class>
                      | <comp value classes> , <value class>
                      ;
<single or set or list> ::= set-of
                         | list-of
                         | <null>
                         ;
<cardinality> ::= [ <min> , <max> ]
                | [ <min> , ]
                | <null>
                ;
```

```
<simple default> ::= DEFAULT : <default value>
                   ;
<default value> ::= <text> | <number>
                  ;
<tuple default> ::= DEFAULT : ( <comp defaults> )
                  | <null>
                  ;
<comp defaults> ::= <default value>
                  | <null>
                  | <comp defaults> , <default value>
                  | <comp defaults> , <null>
                  ;
<comp null rule> ::= not null : <comp attribute list>
                   | <null>
                   ;
<attribute description> ::= DESCRIPTION : <text>
                          | <null>
                          ;
<attribute example> ::= EXAMPLE : <text>
                      | <null>
                      ;
<inverse constraint> ::= inverse of <class name> . <simple attribute name>
                       | <null>
                       ;
<component inverse constraints> ::= <comp inverse constraint>
                                  | <null>
                                  ;
<comp inverse constraint> ::= <comp attribute name> inverse of
                              <class name> . <simple attribute name>
                            | <comp inverse constraint> ,
                              <comp attribute name> inverse of
                              <class name> . <simple attribute name>
                            ;
<simple order by> ::= ORDER BY <attribute name> <ascending_descending>
                    | ORDER BY <attribute name> . <attribute name>
                               <ascending_descending>
                    | ORDER BY <attribute name> [ <class name> ]
                               <attribute name> <ascending_descending>
                    | <null>
                    ;
<tuple order by> ::= ORDER BY <orderby attribute list>
                   | <null>
```

```
                   ;
<orderby attribute list> ::= <component order by>
                           | <orderby attribute list> , <component order by>
                           ;
<component order by> ::= <comp attribute name> <ascending_descending>
                       | <comp attribute name> . <attribute name>
                         <ascending_descending>
                       | <comp attribute name> [ <class name> ]
                         <attribute name> <ascending_descending>
                       ;
<value class> ::= <controlled value class name>
                | <primitive value class>
                | <named primitive value class>
                | <abstract class list>
                | CLASSES
                ;
<primitive value class> ::= BOOLEAN
                          | BINARY '(' <integer> ')'
                          | VARBINARY '(' <integer> ')'
                          | CHAR '(' <integer> ')'
                          | VARCHAR '(' <integer> ')'
                          | INTEGER
                          | SMALLINT
                          | TINYINT
                          | REAL
                          | FLOAT | FLOAT ( <integer> )
                          | DECIMAL | DECIMAL '(' <integer> ')'
                          | DECIMAL '(' ( <integer> ',' <integer> ')'
                          | NUMERIC | NUMERIC '(' <integer> ')'
                          | NUMERIC '(' ( <integer> ',' <integer> ')'
                          | DATETIME
                          | SMALLDATETIME
                          | TIMESTAMP
                          | MONEY
                          | TEXT
                          | IMAGE
                          ;
<abstract class list> ::= <class name list>
                        | specific <class name list>
                        ;
<class name list> ::= <class name>
                    | <class name list> or <class name>
```

```
                    ;
<system metaclass> ::= DATABASES
                     | CLASSES
                     | SCLASSES
                     | SPROTOCOLS
                     ;
<object class name> ::= <class name>
                      ;
<class description> ::= DESCRIPTION : <text>
                      | <null>
                      ;
<class example> ::= EXAMPLE : <text>
                  | <null>
                  ;
<identifier> ::= <attribute list>
               | ( <comp attribute list> )
               ;
<attribute list> ::= <attribute name>
                   | <attribute list> , <attribute name>
                   ;
<represented by> ::= <null>
                   | REP : <attribute list>
                   ;
<generic object classes> ::= <isa> <object class name>
                           | <generic object classes>, <isa> <object class name>
                           ;
<isa> ::= isa | isa*
        ;
<foreign info> ::= <foreign db name or null> <foreign class name>
                 ;
<foreign db name or null> ::= DATABASE: <foreign database name>
                            | <null>
                            ;
<foreign class name> ::= ALIAS: <text>
                       | <null>
                       ;
<protocol class> ::= PROTOCOL CLASS <protocol class name>
                                    <class description>
                                    <class example>
                                    <domain-specific properties>
                                    <protocol specification>
                                    <protocol class delete rules>
```

```
                                  <foreign info>
                   ;
<protocol class name> ::= <class name>
                        ;
<protocol specification> ::= <protocol identifier>
                             <represented by>
                           | <protocol identifier>
                             <represented by>
                             <protocol expansion>
                           ;
<protocol identifier> ::= ID : <attribute list>
                        | ID : ( <comp attribute list > )
                        | <null>
                        ;
<protocol expansion> ::= EXPANSION : <expansion>
                       ;
<expansion> ::= <protocol class name>
              | ( <expansion> )
              | [ <expansion> ]
              | <expansion> , <expansion>
              | <expansion> or <expansion>
              ;
<input or output attribute> ::= ATTRIBUTE <simple attribute name> :
                                <single or set or list>
                                <cardinality> <value class>
                                <simple default>
                                <inverse constraint>
                                <input or output statement>
                                <simple ordered by>
                                <attribute delete rule>
                                <attribute description>
                                <attribute example>
                                <domain-specific properties>
                              ;
<input or output statement> ::= input <input statement>
                              | output <output statement>
                              ;
<input statement> ::= isa <protocol_attribute>
                    | from <from statement>
                    | isa <protocol_attribute> or from <from statement>
                    | <null>
                    ;
```

```
<output statement> ::= isa <protocol_attribute>
                     | <null>
                     ;
<protocol_attribute> ::= <protocol class name> . <simple attribute name>
                       ;
<from statement> ::= <protocol class name> via <via statement>
                   | <from statement> or
                     <protocol class name> via <via statement>
                   ;
<via statement> ::= <attribute name>
                  | <via statement> , <attribute name>
                  ;
<derived attribute> ::= ATTRIBUTE <simple attribute name>
                        DERIVATION : <simple attribute derivation>
                        <attribute description>
                        <attribute example>
                        <domain-specific properties>
                      | ATTRIBUTE <tuple attribute name>
                        ( <comp attribute list> )
                        DERIVATION : userspecified
                        <attribute description>
                        <attribute example>
                        <domain-specific properties>
                      ;
<simple attribute derivation> ::= = <arithmetic expression>
                                | <aggregate expression>
                                | <attribute composition>
                                | userspecified
                                ;
<arithmetic expression> ::= <constant>
                          | <attribute name>
                          | ( <arithmetic expression> )
                          | <arithmetic expression> <op> <arithmetic expression>
                          ;
<op> ::=  + | - | * | /
       ;
<aggregate expression> ::= <aggregate function> of <attribute name>
                         ;
<aggregate function> ::= count | sum | avg | max | min
                       ;
<attribute composition> ::= <composition path>
                          | <attribute composition> or <composition path>
```

```
                          ;
<composition path> ::= <composition element> [ <class name> ]
                     | <attribute name> '.'
                     | <composition path> <composition element> [ <class name> \
]
                     | <composition path> <attribute name>
                     ;
<composition element> ::=   <attribute name>
                        | ! <attribute name>
                        ;
<object class delete rules> ::= <input_for delete rule>
                                <output_of delete rule>
                              ;
<input_for delete rule> ::= INPUT_FOR DELETE NULLIFIES
                          | <null>
                          ;
<output_of delete rule> ::= OUTPUT_OF DELETE <cascades or nullifies>
                          | <null>
                          ;
<protocol class delete rules> ::= <generic_protocol delete rule>
                                  <subprotocols delete rule>
                                  <predecessor_protocols delete rule>
                                  <successor_protocols delete rule>
                                ;
<generic_protocol delete rule> ::= GENERIC_PROTOCOL DELETE
                                            <cascades or nullifies or restricted>
                                 | <null>
                                 ;
<subprotocols delete rule> ::= SUBPROTOCOLS DELETE
                                            <cascades or nullifies or restricted>
                             | <null>
                             ;
<predecessor_protocols delete rule> ::= PREDECESSOR_PROTOCOLS DELETE
                                            <cascades or nullifies or restricted>
                                      | <null>
                                      ;
<successor_protocols delete rule> ::= SUCCESSOR_PROTOCOLS DELETE
                                            <cascades or nullifies or restricted>
                                    | <null>
                                    ;
<attribute delete rule> ::= DELETE <cascades or nullifies or restricted>
                          | <null>
```

```
                          ;
<cascades or nullifies or restricted> ::= RESTRICTED
                                        | <cascades or nullifies>
                                        ;
<cascades or nullifies> ::= CASCADES
                          | NULLIFIES
                          ;
<versioned attributes> ::= <null>
                         | VERSIONED <object attributes>
                         ;
<versioned protocol attributes> ::= <null>
                                  | VERSIONED <protocol attributes>
                                  ;
<domain-specific properties> ::= PROPERTIES : <prop list>
                               | <null>
                               ;
<prop list> ::= <text> <text>
              | <prop list> , <text> <text>
              ;

<OPM derived class> ::= <specialization derived class>
                      | <generalization derived class>
                      ;
<specialization derived class> ::= DERIVED OBJECT CLASS <class name>
                                   <class description>
                                   <class example>
                                   <specialization derivation>
                                   <conditions or null>
                                   <derived view attributes or null>
                                 ;
<specialization derivation> ::= DERIVATION ':' subclass of <class name list>
                              ;
<class name list> ::= <class name>
                    | <class name list> ',' <class name>
                    ;
<conditions or null> ::= <null>
                       | CONDITION ':' <conditions>
                       ;
<derived view attributes or null> ::= <null>
                                    | <derived view attributes>
                                    ;
<conditions> ::= <condition>
```

```
               | <conditions> AND <conditions>
               ;
<condition> ::= NOT IN <class name>
              | <class attribute name> IS NULL
              | <class attribute name> IS NOT NULL
              | <class attribute name> <comp op> <a primitive value>
              | <class attribute name> <comp op> <set of values>
              | <class attribute name> <comp op> <class attribute name>
              | <class attribute name> <match op> <string>
              | <class attribute name> <in op> <set of values>
              | <class attribute name> <in op> <class attribute name>
              ;
<class attribute name> ::= '[' <class name> ']' <attribute name>
                         ;
<comp op> ::= '=' <any all> | '!=' <any all> | '>' <any all>
            | '>=' <any all> | '<' <any all> | '<=' <any all>
            ;
<any all> ::= <null> | ANY | ALL
            ;
<match op> ::= MATCH | NOT MATCH
             ;
<in op> ::= IN | NOT IN
          ;
<set of values> ::= '{' <set of numbers> '}'
                  | '{' <set of strings> '}'
                  ;
<set of numbers> ::= <number>
                   | <set of numbers> ',' <number>
                   ;
<set of strings> ::= <string>
                   | <set of strings> ',' <string>
                   ;
<generalization derived class> ::= DERIVED OBJECT CLASS <class name>
                                   <class description>
                                   <class example>
                                   <generalization derivation>
                                   <derived view attributes>
                                 ;
<generalization derivation> ::= DERIVATION ':' <generalization class list>
                              ;
<generalization class list> ::= <class name>
                              | <generalization class list> or <class name>
```

```
                               ;
<derived view attribute> ::= ATTRIBUTE <simple attribute name>
                             DERIVATION ':' <view simple attribute derivation>
                             <attribute description>
                             <attribute example>
                             <domain-specific properties>
                           ;
<view simple attribute derivation> ::= <extended attribute composition>
                                     | userspecified
                                     ;
<extended attribute composition> ::= <extended composition path>
                                   | <extended attribute composition> or
                                     <extended composition path>
                                   ;
<extended composition path> ::= '[' <class name> ']' <compoaition element>
                              | <extended composition path>
                                '[' <class name> ']' <composition element>
                              ;
<composition element> ::= <attribute name>
                        | '!' <attribute name>
                        ;

<attribute name> ::= /* string */ ;
<class name> ::= /* string */ ;
<database name> ::= /* string */
<integer> ::= /* positive integer */ ;
<constant> ::= /* string with double quotes */ ;
<number> ::= /* positive number or zero */ ;
<text> ::= /* double quoted text */ ;
<code> ::= /* alphanumeric characters or special symbols : + - * / \ ! = . */ ;
<string> ::= /* character string */ ;
<min> ::= /* a positive integer or zero */ ;
<max> ::= /* a positive integer */ ;
```

□

A.2 Additional Constraints for OPM Schemas

1. Metaclasses

   There are four system defined metaclasses: DATABASES, CLASSES, SCLASSES, and SPROTOCOLS. Users are not allowed to change the content of these metaclasses.

2. Controlled Value Classes

   A controlled value class either consists of enumerated atomic string values or numeric ranges, but not both. A range is either a number or an interval specified by lower and upper bounds.

3. Classes

   (a) There are six mutually exclusive types of classes:

      i. primitive value classes (e.g., INTEGER, CHAR(n), etc.);
      ii. controlled value classes;
      iii. base object classes;
      iv. specialization object classes (subclasses of base object classes);
      v. atomic (non-expandable) protocol classes; and
      vi. expandable protocol classes.

   (b) Class names are globally distinct.

   (c) Superclasses of a specialization object class must be base object classes or other specialization object classes.

   (d) Each specialization hierarchy forms a directed acyclic graph.

   (e) Base object classes and protocol classes can be associated with identifier attributes.

   (f) Specialization object classes can either have explicitly specified identifier attributes, or inherit the identifiers from their generic (super) classes.

   (g) If a subclass O_i is specified as: `isa*` O_j, then O_i cannot have common instances with other subclasses of O_j (that are not subclasses of O_i).

   (h) A protocol expansion consists of only atomic protocol classes and/or expandable protocol classes.

   (i) A protocol expansion graph is acyclic, that is, if a protocol class O_i is involved in the expansion of protocol class O_j, then O_j cannot be involved in the expansion O_i or any sub-protocols of O_i.

4. Attributes

(a) All local attributes of a class must have distinct names.

(b) User specified object or protocol attribute names cannot be identical to `OWNER` or `STATUS` attribute names.

(c) Inherited attributes in a subclass are overridden by local attributes with the same names.

(d) The name for a tuple tuple attribute is optional. However, each component attribute in the tuple tuple must have a distinct name.

(e) Nested tuple tuple attributes are not allowed.

(f) A simple or component attribute can only be specified as the inverse of another non-derived simple (single-valued or set-valued) attribute.

(g) Attributes with list of values cannot be associated with inverse constraints.

5. Value Classes

(a) Each attribute must be associated with a value class.

(b) A value class of a simple or component attribute can be of one of the following six types:

i. a controlled value class;

ii. a primitive value class;

iii. a user defined type (or named primitive value class);

iv. a single object class, or a union of several object classes;

v. a single protocol class, or a union of several protocol classes;

vi. CLASSES system metaclass.

6. Identifiers

(a) A specialization class can either have an explicitly defined identifier, or inherit identifier(s) from its superclass(es).

(b) An identifier can consist of one or more simple, tuple or component attributes.

(c) Identifier attributes must be specified as not null, and single-valued.

(d) If an attribute identifier of class O_i is associated with another class O_j as its value class, then O_i is said to be **id-dependent** on O_j. A class O_i cannot be id-dependent on another class O_j if O_j is (transitively) id-dependent on O_i, that is, id-dependency graphs must be acyclic.

(e) Identifier attributes cannot be specified as input, output or derived attributes.

(f) Either an entire tuple attribute is an object identifier, or none of the component attributes in the tuple are part of an identifier.

7. Representative Attributes

(a) Each class can have a set of simple or component attributes to be specified as its representative (`REP`) attributes.

(b) A specialization class will inherit `REP` attributes from its superclasses if this class does not have explicitly specified `REP` attributes.

(c) Each `REP` attribute must be single-valued.

8. Input and Output Attributes

(a) Input and output attributes can be associated only with protocol classes.

(b) Input or output attributes must be simple attributes that are not specified as attribute identifiers nor as derived attributes.

(c) If a protocol P is specified as expanded into several sub-protocols, then the input attributes of P must be subsets of input attributes of the sub-protocols, and the output attributes of P must be subsets of output attributes of the sub-protocols. (Some input/output attributes can be hidden from the higher level protocol.)

(d) Input and output attributes of sub-protocols are related to input and output attributes of higher level protocol using *input is-a* ..., and *output is-a* ... statements.

(e) Input and output connections between sub-protocols are specified using *input from* ... *via* ... statements. If attribute A is an input attribute of protocol class P_i, and the specification for A contains an input statement of the form

$$\textit{input from } P_j \textit{ via } B_1,\ B_2,\ \ldots,\ B_n$$

then

i. If P_i is a sub-protocol of protocol class P_k involved in the expansion of P_k, then P_j must also be a sub-protocol of P_k involved in the same protocol expansion, and P_j must immediately precede P_i in the protocol expansion.

ii. B_1 must be an output attribute of P_j.

iii. For every B_m, $2 \leq m \leq n$: B_m must be a simple or component attribute of class O_m, where O_m is a value class of attribute B_{m-1}.

9. Derived Attributes

Derived simple attributes can be defined using one and only one of the following derivation rules:

(a) attribute composition;

(b) arithmetic expression;

(c) aggregate functions; or

(d) user specified.

10. Arithmetic Expression or Aggregate Function Derived Attributes

(a) A derived attribute A of object or protocol O_i that is specified using an arithmetic expression or an aggregate function, must be a simple, non-identifier attribute. Additionally, if O_i is a protocol class, then A cannot be specified as an input or output attribute of O_i.

(b) An arithmetic expression involved in the specification for derived attribute A of object or protocol class O_i can involve operators, constants, and other single-valued non-derived numeric (simple or component) attributes of O_i.

(c) An aggregate function involved in the specification for derived attribute A of object or protocol class O_i can involve only multi-valued non-derived (simple or component) attributes of O_i, or composition path derived attributes of O_i. Aggregate functions min, max, sum and average can be applied only to numeric attributes (i.e., attributes associated with primitive numeric value classes).

11. Attribute Composition Derived Attributes

(a) Only locally defined (i.e., not inherited), non-derived, simple or component attributes can be involved in attribute compositions.

(b) A simple attribute A of object or protocol class O_i can be associated with an attribute composition derivation consisting of one or (union of) several paths specifications of the following form:

$B_1\ [O_{j_1}]\ B_2\ [O_{j_2}]\ \ldots\ B_n\ [O_{j_n}]$, where

i. each O_{j_k} $(1 \leq k \leq n)$ is an object or a protocol class name;

ii. each B_k $(1 \leq k \leq n)$ is either an attribute name or an attribute name preceded by "!" (indicating an attribute inverse), and

- If B_k is the name of an attribute, C, not preceded by "!", then C must be a non-derived, simple or component attribute that is explicitly associated (i.e., not inherited) with (a) O_i if $k = 1$ or (b) $O_{j_{(k-1)}}$ if $k \geq 2$;
  - if C is associated with a value class V_C consisting of a single object or protocol class, O_C, then O_{j_k} is either O_C or a subclass of O_C;

  - if C is associated with a value class consisting of a union of object or protocol classes, then O_{j_k} must be included in this value class.
- If B_k is the name of an attribute, C, preceded by "!", then C must be a non-derived, simple or component attribute that is explicitly associated (i.e., not inherited) with O_{j_k}.
  - if C is associated with a value class V_C consisting of a single object or protocol class, O_C, then (a) if $k = 1$ then $O_i = O_C$; or (b) if $k \geq 2$ then $O_{j_{(k-1)}} = O_C$;
  - if C is associated with a value class consisting of a union of object or protocol classes, then (a) if $k = 1$ then O_i must be included in this value class, or (b) if $k \geq 2$ then $O_{j_{(k-1)}}$ must be included in this value class.

12. Versions

    (a) Identifier attributes cannot be versioned attributes.

    (b) A non-versioned abstract attribute can refer only to generic objects (therefore keyword "specific" cannot be used in value class definition for non-versioned attributes.)

    (c) A non-versioned abstract attribute cannot be specified as an inverse of a versioned attribute; a versioned abstract attribute cannot be specified as an inverse of another attribute.

    (d) Derivation for non-versioned arithmetic or aggregate function derived attributes can consists of only non-versioned local attributes, and derivation for versioned arithmetic or aggregate function derived attributes can consist of only versioned local attributes.

13. Permissions

    (a) If a schema supports permission modeling, then both `OWNER` and `STATUS` attributes must be specified.

    (b) The `OWNER` attribute is a single-valued, abstract, non-null attribute that can be associated with an optional attribute description, an optional attribute example specification, and optional domain-specific properties.

    (c) The `STATUS` attribute is a single-valued, controlled or primitive, non-null attribute that can be associated with optional attribute default, description, example specification and domain-specific properties. The `STATUS` attribute must be associated with *integer* base data type; an object is considered as *released* if its status value is greater than 0, otherwise it is *non-released*.

(d) `OWNER` attribute and `STATUS` attributes must have distinct names; they cannot be the same as any user-specified attribute names defined in this schema.

(e) All object classes belonging to the same class hierarchy must all be specified with permissions on or off.

(f) Attribute inverse constraints are not allowed in a schema that supports permission modeling.

B The Syntactic Definition for Relational Schemas

The BNF syntax for abstract relational schemas is given below.

```
<relational schema specification> ::= <number of relations>
                                      <db name> <relational schema>
                                    | <db name> <relational schema>
                                    ;
<db name> ::= DATABASE <database name>
            | <null>
            ;
<relational schema> ::= <relation or user specified domain>
                      | <relational schema> <relation or user specified domain>
                      ;
<relation or user specified domain> ::= <relation>
                                      | <view>
                                      | <user specified domain>
                                      ;
<relation> ::= <rel keyword> <relation name>
                     ( <attributes>
                       <primary-key>
                       <alternate-keys>
                       <foreign-keys>
                       &lt;segment&gt;
                       <indexes or null> )
              ;
<rel keyword> ::= RELATION
                | DOMAIN RELATION
                ;
<attributes> ::= <attribute>
               | <attributes> , <attribute>
               ;
<attribute> ::= <attribute name> <attribute domain> <null rule>
                <attribute default>
              ;
<attribute domain> ::= <system domain>
                     | <user specified domain name>
                     ;
<system domain> ::= _id
                  | <data type>
                  | <data type> ( <integer> )
                  | <data type> ( <integer> , <integer> )
                  ;
```

```
<user specified domain> ::= DOMAIN <user specified domain name> <system domain>
                                   <code type>
                                   <values> <default> <in relation or with rule>
                          ;
<code type> ::= <system domain>
              | <null>
              ;
<values> ::= VALUES: <value list>
           | <null>
           ;
<default> ::= DEFAULT: <default value>
            | <null>
            ;
<in relation or with rule> ::= IN RELATION <relation name>
                             | WITH <rule name>
                             ;
<value-list> ::= <enumerated values>
               | <ranges>
               ;
<enumerated values> ::= <value_code_desc>
                      | <enumerated values> , <value_code_desc>
                      ;
<value_code_desc> ::= <value>
                    | ( <value> , <code> )
                    | ( <value> , <code> , <value description> )
                    ;
<value description> ::= <text>
                      ;
<ranges> ::= <range>
           | <ranges> , <range>
           ;
<range> ::= <number> - <number>
          ;
<null rule> ::= NO NULLS
              | NULLS ALLOWED
              | <null>
              ;
<attribute default> ::= DEFAULT <real>
                      | DEFAULT <text>
                      | <null>
                      ;
<primary-key> ::= PRIMARY KEY ( <attribute list> )
```

```
                ;
<alternate-keys> ::= <alternate-key list>
                   | <null>
                   ;
<alternate-key list> ::= <alternate-key>
                       | <alternate-key list> <alternate-key>
                       ;
<alternate-key> ::= ALTERNATE KEY ( <integer> ) ( <attribute list> )
                  ;
<foreign-keys> ::= <foreign-key list>
                 | <null>
                 ;
<foreign-key list> ::= <foreign-key>
                     | <foreign-key list> <foreign-key>
                     ;
<foreign-key> ::= FOREIGN KEY ( <attribute list> )
                    REFERENCES <referenced relations> delete-rule
                    <mutual exclusions>
                ;
<referenced relations> ::= <relation name>
                      | <referenced relations> OR <relation name>
                      ;
<mutual exclusions> ::= <null>
                      | NOT IN <excluded relations>
                      ;
<excluded relations> ::= <relation name>
                       | <excluded relations> AND <relation name>
                       ;
<attribute list> ::= <attribute name>
               | <attribute list> , <attribute name>
               ;
<delete-rule> ::= RESTRICTED | CASCADE | NULLIFY
                ;
&lt;segment&gt; ::= SEGMENT &lt;segment name&gt;
            | <null>
            ;
<indexes or null> ::= <indexes>
                    | <null>
                    ;
<indexes> ::= <index>
            | <indexes> <index>
            ;
```

```
<index> ::= INDEX <text>
          ;
<view> ::= VIEW <view name>
           DEFINITION : <SQL view definition> ;
         ;
<attribute name> ::= /* character string */
                   ;
<relation name> ::= /* character string */
                  ;
<view name> ::= /* character string */
              ;
<rule name> ::= /* character string */
              ;
<user specified domain name> ::= /* character string */
                               ;
<data type> ::= /* character string */
              ;
&lt;segment name&gt; ::= /* character string */
                 ;
<integer> ::= /* positive integer */
            ;
<real> ::= /* real number */
         ;
<number of relations> ::= <integer>
                        ;
<text> ::= /* quoted text */
         ;
<SQL view definition> ::= /* SQL view definition statement */
                  ;
<value> ::= /* a numeric value or an alphanumeric value within quotes */
          ;
```

□