Attribute Inheritance Implemented on Top of a Relational Database System

Stefan Böttcher
IBM Deutschland GmbH
Scientific Center
Institute for Knowledge Based Systems
Postfach 80 08 80
D-7000 Stuttgart 80
West Germany

Abstract

We present an implementation technique which solves integrity checking, query evaluation and transaction synchronization tasks in database systems with multiple attribute inheritance and which uses the support of a relational database system. The basic idea is to map arbitrary integrity constraints, queries and locks associated with classes of an inheritance lattice onto integrity constraints, queries and locks associated with relations of an underlying relational database.

1 Introduction

Some of the major requirements for database systems with attribute inheritance [Andrews87], [Batory88a], [Batory88b], [Bloom87], [Mylopoulos80], [Mylopoulos86] are to support not only the inheritance of attributes including their type restriction, but also to support the inheritance of arbitrary constraints associated with the objects of certain classes. This includes that the database system shall support referential contraints connecting two or more classes. Further, the system shall support transaction synchronization, e.g. queries on a class c_1 have to be synchronized against write operations affecting elements of some subclass c_2 of class c_1.

In this paper we present an implementation technique supporting these requirements. Arbitrary read operations on classes of an inheritance lattice are implemented by read operations on database views. Write operations on classes (i.e. NEW, DISPOSE and set oriented write operations) are implemented by write operations on database relations. We further show how to implement an integrity checker and a transaction synchronization component of a database system with attribute inheritance (DBSAI). Different from the papers cited above, we show how operations on arbitrary predicatively described subsets of classes can be mapped to a relational DBS.

The next section defines a DBSAI and a set of basic operations on it. The third section describes a mapping of an inheritance lattice onto a relational database containing fact relations and non-recursive views. We call this mapping *Canonical Mapping*. In the fourth section, we show how all the operations defined are implemented using Canonical Mapping, and further, how integrity checking and transaction synchronization tasks on arbitrary inheritance lattices can be solved by propagating them to relational database system tasks.

2 Database systems with Attribute Inheritance

The objective of this section is to specify what shall be implemented: we define the state of a database with attribute inheritance and the semantics of the elementary write operations NEW and DISPOSE. We further introduce queries, set oriented write operations, integrity constraints and a concept of predicate locks for DBSAI. Examples are given in section 3.1.

2.1 The state of a database with attribute inheritance

In order to define the semantics of the DBSAI write operations NEW and DISPOSE, we define a state of a database with attribute inheritance. A *state* of a database with attribute inheritance is defined as a tuple

(O, D, A, C, IS-A, IN, IS-ATTRIBUTE-OF, IC)

with the following properties.

O is a set of objects (denoted as o_i).

D is a set of domains d_i, which are ordered sets of values. Typical domains considered are integers, strings, etc.

A is a linear ordered set of partial functions a_i (called attributes) mapping objects to values of a certain domain $d_i \in D$.

C is a set of classes with a partial order IS-A, called the subclass order. Whenever (c_i IS-A c_j), we call c_i the subclass of c_j and c_j the superclass of c_i.

IS-ATTRIBUTE-OF is a binary relation describing which attributes are defined for which class, i.e. (a_i IS-ATTRIBUTE-OF c_i) denotes that the attribute a_i is defined for (all objects of) the class c_i. As in TAXIS [Mylopoulos80], we require that every attribute defined for a class c_i is also defined for all subclasses c_j of c_i, i.e. if (a_i IS-ATTRIBUTE-OF c_i) and (c_j IS-A c_i) then (a_i IS-ATTRIBUTE-OF c_j). Further, we apply the linear order of attributes to the set of attributes defined for a class c_i.

IN is a containment relation, i.e. (o_i IN c_i) denotes that the object o_i is an instance of class c_i. The containment relation IN has to fulfill the following conditions:

- Every object belongs to at least one class, i.e. for all $o_i \in O$ there exists $c_i \in C$ such that (o_i IN c_i).
- Every object contained in a class c_i is also contained in every superclass of c_i, i.e. if (o_i IN c_i) and (c_i IS-A c_j) then (o_i IN c_j).
- If (o_i IN c_i), then for every attribute a_i of c_i, $a_i(o_i)$ is defined.

IC is a set of integrity constraint definitions. The set of possible integrity constraint definitions is characterized by the syntax for integrity constraint definitions given in appendix A.

2.2 The write operations NEW and DISPOSE

We define a write operation on a database with attribute inheritance as the process of transforming one state into another state. Throughout this paper we discuss the implementation of two elementary write operations, NEW and DISPOSE. Both operations are only partially defined, i.e. before they are executed certain preconditions have to be fulfilled. We describe the semantics of both operations by giving their preconditions and state transformations.

Given a state

(O, D, A, C, IS-A, IN, IS-ATTRIBUTE-OF, IC) ,

the preconditions of the operation NEW (c_i, o_i, $(v_1,\dots,v_n)$) are the following:

1. the object to be created does not exist already, i.e. $o_i \notin O$,
2. the class c_i has n attributes, say a_j, and for each one the range d_j of the function a_j includes the value v_j.

If the preconditions are true, the execution of the NEW operation transforms the given DBSAI state into a state

(O', D, A', C, IS-A, IN', IS-ATTRIBUTE-OF, IC)

with

$O' =_{def} O \cup \{ o_i \}$ and

$IN' =_{def} IN \cup \{ (o_i , c_j) \mid c_i = c_j \vee c_i \text{ IS-A } c_j \}$ and

A' can be derived from A by augmenting every attribute a_k of c_i by a partial definition which maps o_i onto the value v_k given for the attribute a_k.

Note that by this definition an object created in class c_i becomes an object of all superclasses c_j of c_i as well.

The DISPOSE operation can be defined similarly. Given a state

(O, D, A, C, IS-A, IN, IS-ATTRIBUTE-OF, IC) ,

the precondition of the operation DISPOSE (c_i , o_i) is the following: The object to be disposed from class c_i must exist in c_i, i.e. (o_i IN c_i) must be true. If the precondition is true, the execution of the DISPOSE operation transforms the given DBSAI state into a state

(O', D, A', C, IS-A, IN', IS-ATTRIBUTE-OF, IC)

with

$O' =_{def} O \setminus \{ o_i \}$ and

$IN' =_{def} IN \setminus \{ (o_i , c_j) \mid c_j \in C \}$ and

A' can be derived from A by removing the pair $(o_i, a_k(o_i))$ from every attribute a_k of c_i.

The NEW operation requires to specify the smallest class in which an object has to be created, while it is sufficent to specify any superclass within the DISPOSE operation. This simplifies the definition of set oriented write operations, which are defined in section 2.4.

2.3 Set valued queries and boolean queries

In order to define a query language we modify the relational calculus in such a way that the query expressions contain class names instead of relation names. Our query language syntax is similar to that of the relational database programming language Pascal/R [Schmidt77]. A complete definition of our query language syntax is given in appendix A. For example, the query expression

{ EACH o_i IN c_i (SOME o_j IN c_j
(o_i.attribute$_1$ = o_j.attribute$_2$)) }

denotes the set of all objects o_i in class c_i for which there exists an object o_j in class c_j so that the value for attribute$_1$

of o_i is equal to the value for attribute_2 of o_j. We call this query expression a set valued query expression.

In addition to set valued queries, the query language supports boolean valued queries. This includes nested quantified queries, e.g. "is there some object o_i in class c_i so that its attribute attribute_1 is less or equal to the attribute attribute_2 of every object o_2 of class c_2", which is described by the following expression

$$\text{SOME } o_i \text{ IN } c_i \text{ (ALL } o_j \text{ IN } c_j \text{ (} o_i.\text{attribute}_1 \leq o_j.\text{attribute}_2 \text{)) .}$$

The syntax of the query language specified in appendix A also allows for more complex nested queries.

2.4 Set oriented write operations

Set oriented write operations can be defined using set valued query expressions and the operations NEW and DISPOSE.

If c_i is a class and S is a set valued query expression, then set oriented write operations deleting (:–) a set S from a class c_i or inserting (:+) it into a class c_i,

c_i :– S and
c_i :+ S ,

can be defined using the definitions of NEW and DISPOSE. The set oriented delete statement

c_i :– S

can be defined by applying the operation DISPOSE (c_i, o_i) to every object o_i selected by the query expression S. The set oriented insert statement

c_i :+ S

inserting a set S to a class c_i, can be defined similar by applying the operation NEW (c_i , o_i , $(v_{i_1},...,v_{i_n})$) to every element $(v_{i_1},...,v_{i_n})$ selected by the query expression S.

2.5 Integrity constraints

Throughout this paper we deal with the following set of integrity constraints. We will show how to implement arbitrary integrity constraints that can be expressed by boolean queries. To keep the notation simple, we express integrity constraints in the same way as we specify boolean queries of the query language, e.g. a referential integrity constraint is expressed as

$$\text{CONSTRAINT ALL } o_i \text{ IN } c_i \text{ (SOME } o_j \text{ IN } c_j \text{ (} o_i.\text{attribute}_1 = o_j.\text{attribute}_2 \text{)) .}$$

Note that the integrity constraints described here are more general than type constraints since referential integrity constraints relate the elements of two classes to each other. Type constraints are only a special kind of integrity constraints expressible in our query language.

2.6 Transaction synchronization using predicate locks

Conceptual modelling languages like TAXIS [Mylopoulos80] have a concept of transactions which operate on classes. The DBSAI has to synchronize read and write operations of these transactions, e.g. queries on a class c_i have to be synchronized against write operations on some subclass of c_i. To support these synchronization tasks we implement a concept of locks. We decided to implement predicate locks to avoid the "phantom problem" [Eswaran76]. When these predicate locks are assigned to classes they also block some operations on subclasses. For example, a predicate lock

$$\text{READLOCK } \{ \text{ EACH } o_i \text{ IN } c_i \text{ (} o_i.\ \text{attribute}_1 \leq 1000 \text{) } \}$$

blocks write operations of parallel transactions on such objects of class c_i that have an attribute_1 value less or equal to 1000. Note that the lock operation includes objects of subclasses of c_i, that have an attribute_1 value less or equal to 1000. Hence, write operations of parallel transactions on these objects are blocked too.

The implementation technique presented supports (predicative) read and write locks on subsets of a class. Lock statements can lock arbitrary subsets of a class, where the size of the subsets can be characterized by set valued query expressions.[1] Note that this approach is different from the approach of [Garza88], where predicate locks are not supported. In the fourth section, we show that an efficient implementation of predicate locks (e.g. [Böttcher86]) can also be adapted to DBSAI.

Having specified what to implement, we now present a technique how to implement these tasks which is based on a relational database system. For this purpose we define a function which maps all the tasks defined for DBSAI onto tasks for a relational DBS.

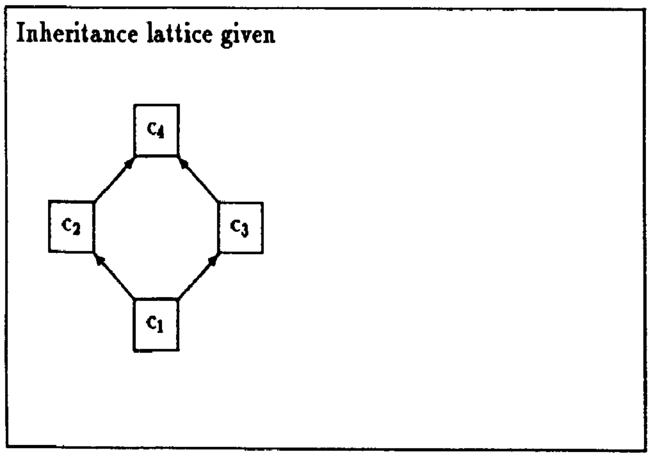
3 Canonical Mapping

In what follows, we define a function which maps each class of an inheritance lattice onto a pair consisting of a view *and* an updatable relation of a relational database. This function is called *Canonical Mapping*. It is used for the implementation of arbitrary read and write operations, integrity checks and lock operations. Before we outline a complete set of definitions, we shall motivate them by giving the following example.

[1]The syntax of lock statements is defined in appendix A.

3.1 A motivating example

We want to implement arbitrary read and write operations on a multiple inheritance lattice with classes c_1, c_2, c_3, and c_4, where c_1 is a subclass of c_2 and c_3, and c_2 and c_3 are subclasses of c_4.



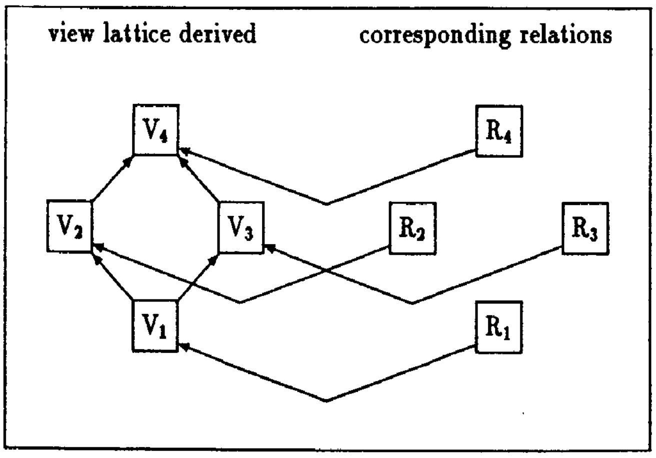
$\boxed{c_1} \longrightarrow \boxed{c_2}$ means c_1 is a subclass of c_2

Canonical Mapping means the following: Each class c_i is associated with a corresponding database relation R_i and a view V_i. Each attribute of c_i is associated with a corresponding attribute of R_i and a corresponding attribute of V_i, i.e. both R_i and V_i by definition have the same set of attributes as c_i. Every read operation on class c_i will be implemented by a read operation on the view V_i. However, every NEW operation applied to class c_i will be implemented by an insert operation on R_i. An implementation of DISPOSE operations will be presented in section 4.2.

The idea of Canonical Mapping is the following: If c_1 is a subclass of c_2, R_1 and R_2 are the corresponding database relations, and V_1 and V_2 are the corresponding views, then R_2 does not contain tuples for entities for which there are tuples in R_1 but the view V_2 does.

In order to derive the view definitions V_i for a given inheritance lattice, we introduce the notation $\pi_j(V_i)$, denoting the projection of V_i on the attributes defined for class c_j. Note that whenever c_i is a subclass of c_j, then all attributes of c_j are attributes of c_i as well. Hence, for this j the projection of V_i on the attributes defined for class c_j, $\pi_j(V_i)$, is well defined.

The following view definitions of the views V_1,...,V_4 can be derived from the lattice structure given in the previous diagram:



$\boxed{V_1} \longrightarrow \boxed{V_2}$ means V_1 is used to define V_2

$V_1 =_{def} R_1$ i.e. V_1 is defined to be equal to R_1

$V_2 =_{def} R_2 \cup \pi_2(V_1)$

$V_3 =_{def} R_3 \cup \pi_3(V_1)$

$V_4 =_{def} R_4 \cup \pi_4(V_2) \cup \pi_4(V_3)$

Given these view definitions we can implement queries on classes by queries on views. We simply implement an arbitrary query ranging over classes c_i by substituting each occurrence of a class c_i with the corresponding view V_i. For example a query

$$\{ \text{EACH } o_2 \text{ IN } c_2 \text{ (} o_2.\text{attribute}_1 \leq 1000 \text{) } \}$$

is translated to

$$\{ \text{EACH } o_2 \text{ IN } V_2 \text{ (} o_2.\text{attribute}_1 \leq 1000 \text{) } \}.$$

This translated query to the database view V_2 can be evaluated by the database system. Note that the answer to the translated query may be changed by a successful NEW operation creating an object of class c_2 [2] (and even by creating an object of class c_1). Since NEW operations which create objects of class c_1 or c_2 may change the result of the query, the query has to be synchronized against NEW operations of parallel transactions. A read lock on class c_2 can be implemented by read locks on both database relations, R_1 and R_2.

3.2 Definition of Canonical Mapping

We are now ready to define Canonical Mapping. Note that the definition is obtained simply by generalizing the rules

[2] The NEW operation which creates an object of class c_2 is implemented by inserting a tuple into R_2.

given in the example.

Canonical Mapping maps every class c_i of a database with attribute inheritance to a pair consisting of a view V_i and a database relation R_i.

The schema definition of R_i can be derived from the domains of the attributes defined for c_i. If $(a_1,...,a_n)$ is an n-tuple of all attributes of c_i and $d_1,...,d_n$ are the domains of these attributes, then the schema of R_i, *schema*(R_i), is defined by

$$schema(R_i) =_{def} d_1 \times ... \times d_n .$$

The view definition of V_i is obtained by the following rule. If $c_{i_1},...,c_{i_n}$ are all subclasses of c_i within a given database state, then V_i is defined by

$$V_i =_{def} R_i \cup \pi_i(V_{i_1}) \cup ... \cup \pi_i(V_{i_n}) .$$

Remember, that $\pi_i(V_{i_j})$ denotes the projection of V_{i_j} onto the attributes defined for the class c_i.

Using this Canonical Mapping we can now implement queries, write operations, integrity constraints and predicate locks associated with arbitrary classes of an inheritance lattice by mapping these tasks onto a relational database system.

4 Implementation of queries, write operations, constraints and predicate locks for DBSAI

4.1 The translation of queries

Given an arbitrary query or subquery containing classes, we can translate it into a query on database relations using the following three-step transformation algorithm. First, every class is substituted by its corresponding view using Canonical Mapping. Second, every view is substituted by its definition. Third, all union operators are eliminated using the following rules as long as they are applicable.

For all set valued expressions R and S and all boolean query expressions $F(o_i)$ we have the following transformation rules[3]:

1. { EACH o_i IN $(R \cup S)$ ($F(o_i)$) } $\rightarrow$
   { EACH o_i IN R ($F(o_i)$) } $\cup$
   { EACH o_i IN S ($F(o_i)$) }
2. SOME o_i IN $(R \cup S)$ ($F(o_i)$) $\rightarrow$
   SOME o_i IN R ($F(o_i)$) OR
   SOME o_i IN S ($F(o_i)$)
3. ALL o_i IN $(R \cup S)$ ($F(o_i)$) $\rightarrow$
   ALL o_i IN R ($F(o_i)$) AND ALL o_i IN S ($F(o_i)$) .

[3]For a definition of the query syntax see appendix A.

After applying this transformation algorithm every query expression contains only database relations.

This transformation algorithm will also be used to solve the following tasks. Integrity constraints given for classes are transformed into integrity tests to be performed on database relations, and locks required on classes are transformed into lock expressions on database relations.

4.2 The implementation of write operations

NEW, DISPOSE and set oriented write operations on a class c_i are implemented by write operations on the corresponding database relation R_i.

An operation NEW (c_i , o_i , $(v_1,...,v_n)$) shall create an object o_i of a class c_i and assign the values $(v_1,...,v_n)$ to $(a_1(o_1),...,a_n(o_n))$. This operation can be implemented by an insert operation, inserting the tuple $(v_1,...,v_n)$ into the database relation R_i. Note that the tuple $(v_1,...,v_n)$ representing the object o_i is visible to every query on V_i after the insertion of $(v_1,...,v_n)$ into R_i.

An operation DISPOSE (c_i , o_i) shall dispose an object o_i of class c_i. Since the object o_i may be an instance of some subclass c_j of c_i too, the object has to be deleted from that relation it has been inserted into. One simple implementation technique is to determine in which database relation the object is represented and to delete it from this relation.

Set oriented write operations can be implemented similar to the NEW and DISPOSE operations. A set oriented insert operation

c_i :+ S

is implemented by a set oriented insert operation on a database relation inserting all elements of S into the database relation R_i, while a set oriented delete operation

c_i :- S

is implemented by deleting all objects described by S from the database relations R_i, R_{i_1}, ..., R_{i_n} corresponding to the class c_i and its subclasses c_{i_1}, ..., c_{i_n} .

4.3 The propagation of integrity constraints

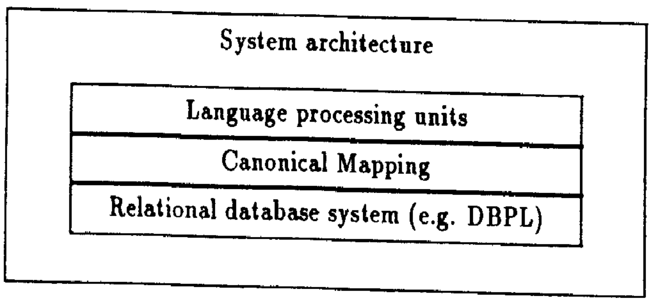
Like boolean queries, integrity constraints ranging over classes can be mapped onto integrity constraints over views. These constraints can be propagated onto the database relations of the underlying database by using the propagation rules listed in section 4.1. After applying these transformation steps we have a set of integrity constraints defined on database relations only, which implement the integrity constraints defined on classes of the given inheritance lattice. Both integrity checking and predicate locks are implemented e.g. in the DBPL database system which was developed at the University of Frankfurt [Böttcher89], [Schmidt88].

4.4 The propagation of predicate locks

Given arbitrary lock requests as described in section 2.5, the set valued query expression of these lock requests can be transformed into set valued query expressions containing only database relations by using the rules given for query transformation in section 4.1. After the transformation a set of lock requests containing only database relations remains. These lock requests can be submitted to an ordinary relational database system. The database system should provide some efficient locking algorithm for predicate locks (e.g. [Böttcher86]) in order to synchronize lock operations of different transactions against each other.

4.5 System architecture

The DBSAI based on Canonical Mapping is implemented by the following three-layered architecture. The top layer is a language processor (an interpreter or a compiler) and processes DBSAI programs. The intermediate layer is the Canonical Mapping layer. It performs the transformation steps described in this paper. Queries, write operations, integrity tests, lock and unlock operations associated with classes are transformed into queries, write operations, integrity tests, lock and unlock operations, which can be executed by a relational database system. The bottom layer contains a relational database system. It executes the tasks it receives from the Canonical Mapping layer.


System architecture
Language processing units
Canonical Mapping
Relational database system (e.g. DBPL)

5 Summary and Conclusion

We have shown that Canonical Mapping is a natural way of mapping a variety of tasks of a DBSAI onto a relational DBS. Hence, using Canonical Mapping, queries, write operations, integrity checks and predicate locks in a DBSAI can be implemented by queries, write operations, integrity checks and predicate locks in a relational DBS. Therefore, Canonical Mapping seems to be a natural way to extend the relational data model with attribute inheritance.

References

[Andrews87] Andrews, T., Harris, C.: Combining Language and Database Advices in an Object-Oriented Development Environment. *OOPSLA Conference Proceedings*, Orlando, Florida, 1987.

[Batory88a] Batory, D.S.: Concepts for a Database System Synthesizer. *ACM PoDS*, 1988.

[Batory88b] Batory, D.S., et al.: Genesis: An Extensible Database Management System. *IEEE ToSE*, November 1988.

[Bloom87] Bloom, T., Zdonik, S.: Issues in the Design of Object-Oriented Database Programming Languages. *OOPSLA Conference Proceedings*, Orlando, Florida, 1987.

[Böttcher86] Böttcher, S., Jarke, M., Schmidt, J.W.: Adaptive Predicate Managers in Database Systems. *Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto, 1986.

[Böttcher89] Böttcher, S.: Prädikative Selektion als Grundlage für Transaktionssynchronisation und Datenintegrität. PhD Thesis, Johann Wolfgang Goethe Universität Frankfurt, 1989.

[Carey88] Carey, M.J., DeWitt, D.J., Vandenberg, S.L.: A Data Model and Query Language for EXODUS. *ACM SIGMOD Int. Conf.*, Chicago, 1988.

[Eswaran76] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database System. *CACM, 19*, 11, 1976.

[Garza88] Garza, J.F., Kim, W.: Transaction Management in an Object-Oriented Database System. *ACM SIGMOD Conf.*, Chicago, 1988.

[Mylopoulos80] Mylopoulos, J., Bernstein, P.A., Wong, H.K.T.: A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems, 5*, 2, 1980, pp. 185-207.

[Mylopoulos86] Mylopoulos, J., Brodie, M.L.(Eds.): On Knowledge Base Management Systems. Springer, 1986.

[Schmidt77] Schmidt, J.W.: Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems, 2*, 3, 1977, pp. 247-261.

[Schmidt88] Schmidt, J.W., Eckhardt, H., Matthes, F.: DBPL Report. DBPL-Memo 111-88, Johann Wolfgang Goethe Universität Frankfurt, 1988.

Appendix A

In this appendix we outline a query language containing both boolean queries and set valued queries. The query language is similar to that of DBPL [Schmidt88] except that here queries range over classes instead of relations. Further we describe statements for write operations, for the definition of integrity constraints and for the synchronization statements lock and unlock.

For the definition of this query language we assume states of a database with attribute inheritance

(O, D, A, C, IS-A, IN, IS-ATTRIBUTE-OF, IC)

to be defined as in section 2. In order to define boolean query expressions we start defining atomic boolean query expressions.

A1 $o_i.a_i$ comp $o_j.a_j$ and

A2 $o_i.a_i$ comp v

are atomic boolean query expressions, where comp$\in\{ = , \leq , \geq , \neq , < , > \}$, v is a value of a domain d_i, i.e. v$\in d_i$ and $d_i \in$D, a_i and a_j are attributes with domain d_i, and $a_i(o_i)$ and $a_j(o_j)$ (denoted as $o_i.a_i$ and $o_j.a_j$) are defined.

Now we can define boolean query expressions and set valued query expressions recursively.

B1 Atomic boolean query expressions are boolean query expressions.

B2 If A and B are boolean query expressions then
A AND B, A OR B, and NOT A are boolean query expressions.

B3 TRUE and FALSE are boolean query expressions.

B4 If B is a boolean query expression, and S is a set valued query expression, then
ALL o_i IN S (B)
SOME o_i IN S (B)
are boolean query expressions.

Set valued query expressions are defined by the following rules.

S1 If c_i is a class (i.e. $c_i \in$C), and B is a boolean query expression, then
{ EACH o_i IN c_i (B) }
is a set valued query expression.

S2 Further, if R and S are set valued expressions, then
π_i(S) and
R $\cup$ S
are set valued expressions, where π_i(S) denotes the projection of S onto the attributes defined for the class $c_i \in$C. We assume that the projection removes duplicates.

S3 To get a relational complete query language, we further define joins: if R and S are set valued expressions and B is a boolean query expression, then
{ EACH o_i IN R $\times$ S (B) }
is a set valued expression. Note however, that this join is not used to define Canonical Mapping.

Set oriented write operations can be described as follows. If c_i is a class and S is a set valued query expression, then

c_i :- S and
c_i :+ S ,

are set oriented write operations. Further, the NEW and DISPOSE statements described in section 2.2 are write operations.

An integrity constraint definition can be described as follows. If B is a boolean valued query, then

CONSTRAINT B

is an integrity constraint definition.

Lock and unlock statements are defined as follows. If S is a set valued query expression, then

READLOCK S and

WRITELOCK S

are legal lock statements and

UNLOCK S

is a legal unlock statement in any transaction.