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 constraints connecting two or more classes. Further, the system shall support transaction synchronization, e.g. queries on a class c2 have to be synchronized against write operations affecting elements of some subclass c1 of class c2.

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.

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 \( c_i \).

\( IS-ATTRIBUTEOF \) is a binary relation describing which attributes are defined for which class, i.e. \( (a_i, IS-ATTRIBUTEOF c_j) \) denotes that the attribute \( a_i \) is defined for all objects of the class \( c_j \). As in TAINS [Mylopoulos80], we require that every attribute defined for a class \( c_j \) is also defined for all subclasses \( c_i \) of \( c_j \), i.e. if \( (a_i, IS-ATTRIBUTEOF c_j) \) and \( (c_j IS-A c_i) \) then \( (a_i, IS-ATTRIBUTEOF c_i) \).

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 \( (o_i, IN c_j) \), 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

\[
(O, D, A, C, IS-A, IN, IS-ATTRIBUTEOF, IC)
\]

the preconditions of the operation NEW \( (c_i, o_i, (\tau_1, ..., \tau_n) ) \) 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 \( \tau_j \).

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

\[
(O', D, A', C, IS-A, IN, IS-ATTRIBUTEOF, IC)
\]

with

\[
O' = def O \cup \{ o_i \}
\]

\[
IN' = def IN \cup \{ (o_i, c_j) \mid c_j = c_i \lor c_i IS-A c_j \}
\]

and

\[
A' \text{ can be derived from } A \text{ by augmenting every attribute } a_i \text{ of } c_i \text{ by a partial definition which maps } o_i \text{ onto the value } \tau_k \text{ given for the attribute } a_k.
\]

Note that by this definition an object created in class \( c_i \) becomes an object of all subclasses \( c_j \) of \( c_i \) as well.

The DISPOSE operation can be defined similarly. Given a state

\[
(O, D, A, C, IS-A, IN, IS-ATTRIBUTEOF, IC)
\]

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

\[
(O', D, A', C, IS-A, IN', IS-ATTRIBUTEOF, IC)
\]

with

\[
O' = def O \setminus \{ o_i \}
\]

\[
IN' = def IN \setminus \{ (o_i, c_j) \mid c_j \in C \}
\]

and

\[
A' \text{ can be derived from } A \text{ by removing the pair } (o_i, a_k(o_i)) \text{ from every attribute } a_k \text{ of } c_i.
\]

The NEW operation requires to specify the smallest class in which an object has to be created, while it is sufficient 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 \(a_i\) is equal to the value for attribute\(j\) of \(a_j\). 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 \(a_i\) in class \(c_i\) so that its attribute attribute\(j\) is less or equal to the attribute attribute\(j\) of every object \(a_j\) of class \(c_j\)" which is described by the following expression

\[
\text{SOME } a_i \text{ IN } c_i \left( \text{ALL } a_j \text{ IN } c_j \left( a_i.\text{attribute}_j \leq a_j.\text{attribute}_j \right) \right).
\]

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 \(c_i\),

\[
\begin{align*}
\text{\(c_i \leftarrow \neg S\)} & \quad \text{and} \\
\text{\(c_i \leftarrow S\)} & \\
\end{align*}
\]

can be defined using the definitions of NEW and DISPOSE.

The set oriented delete statement

\[
\text{\(c_i \leftarrow \neg S\)}
\]

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

\[
\text{\(c_i \leftarrow S\)}
\]

inserting a set \(S\) to a class \(c_i\), can be defined similarly by applying the operation NEW \((c_i, a_i, (v_1, \ldots, v_n))\) to every element \((v_1, \ldots, v_n)\) 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 } \begin{align*}
\text{\(\text{ALL } a_i \text{ IN } c_i \left( \text{SOME } a_j \text{ IN } c_j \left( a_i.\text{attribute}_j = a_j.\text{attribute}_j \right) \right) \).}
\end{align*}
\]

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" [Rewaran76]. When these predicate locks are assigned to classes they also block some operations on subclasses. For example, a predicate lock

\[
\text{READLOCK } \{ \text{EACH } a_i \text{ IN } c_i \left( a_i.\text{attribute}_j \leq 1000 \right) \}
\]

blocks write operations of parallel transactions on such objects of class \(c_i\) that have an attribute value less or equal to 1000. Note that the lock operation includes objects of subclasses of \(c_i\), that have an attribute 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. 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.

\[\text{The syntax of lock statements is defined in appendix A.}\]
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 $c_4$.

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 $c_5$, $R_1$ and $R_5$ are the corresponding database relations, and $V_1$ and $V_5$ are the corresponding views, then $R_5$ does not contain tuples for entities for which there are tuples in $R_1$ but the view $V_5$ 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 $c_i$ are attributes of $c_j$ 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 $V_1$ to $V_4$ can be derived from the lattice structure given in the previous diagram:

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

\[ \{\text{EACH } o_2 \text{ IN } c_5 \ (\ o_2.\text{attribute}_1 \leq 1000 ) \} \]

is translated to

\[ \{\text{EACH } o_2 \text{ IN } V_5 \ (\ o_2.\text{attribute}_1 \leq 1000 ) \} \].

This translated query to the database view $V_5$ 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 $c_2$ \(^7\) 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 $R_5$.

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

\(^7\)The NEW operation which creates an object of class $c_2$ is implemented by inserting a tuple into $R_2$. 

506
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 \( d_1, ..., d_m \) are the domains of these attributes, then the schema of \( R_i \), \( \text{schema}(R_i) \), is defined by

\[
\text{schema}(R_i) = a_1 x ... x a_n.
\]

The view definition of \( V_i \) is obtained by the following rule. If \( c_{i1}, ..., c_{il} \) are all subclasses of \( c_i \) within a given database state, then \( V_i \) is defined by

\[
V_i = \pi_f R_i \cup \pi_l(V_{i1}) \cup ... \cup \pi_l(V_{il}).
\]

Remember, that \( \pi_f(V_{ij}) \) denotes the projection of \( V_{ij} \) 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 \( F(a_i) \) we have the following transformation rules:

1. \( \{ \text{EACH } a_i \in (R \cup S) \; (F(a_i)) \} \rightarrow \{ \text{EACH } a_i \in R \; (F(a_i)) \} \cup \{ \text{EACH } a_i \in S \; (F(a_i)) \} \)
2. \( \{ \text{SOME } a_i \in (R \cup S) \; (F(a_i)) \rightarrow \; \{ \text{SOME } a_i \in R \; (F(a_i)) \} \lor \{ \text{SOME } a_i \in S \; (F(a_i)) \} \}
3. \( \{ \text{ALL } a_i \in (R \cup S) \; (F(a_i)) \rightarrow \; \{ \text{ALL } a_i \in R \; (F(a_i)) \} \land \{ \text{ALL } a_i \in S \; (F(a_i)) \} \}

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 \( (a_1(o_i), ..., a_n(o_i)) \). 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 \( R_i, R_{i1}, ..., R_{in} \) corresponding to the class \( c_i \) and its subclasses \( c_{i1}, ..., c_{in} \).

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.

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


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 \( o;\cdot a; \text{comp} o;\cdot a; \text{and} \)
A₂ \( o;\cdot a; \text{comp} v \)

are atomic boolean query expressions, where \( \text{comp} \in \{ =, \leq, \geq, \neq, <, > \} \), \( v \) is a value of a domain \( d; \), i.e. \( v \in d; \), and \( d; \in D \), \( a; \) and \( a; \) are attributes with domain \( d; \), and \( a; (o;) \) and \( a; (o;) \) (denoted as \( o;\cdot a; \) and \( o;\cdot a; \)) 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
\( A \text{ AND } B, A \text{ OR } B, \text{ and NOT } A \) 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
\( \{ \text{EACH } o; \text{ IN } S \ ( B ) \} \)
\( \{ \text{SOME } o; \text{ IN } S \ ( B ) \} \)
are boolean query expressions.

Set valued query expressions are defined by the following rules.

S₁ If \( c; \) is a class (i.e. \( c; \in C \)), and B is a boolean query expression, then
\( \{ \text{EACH } o; \text{ IN } c; \ ( B ) \} \)
is a set valued query expression.
S₂ Further, if R and S are set valued expressions, then
\( \pi_r(S) \text{ and } R \cup S \)
are set valued expressions, where \( \pi_r(S) \) denotes the projection of \( S \) onto the attributes defined for the class \( c; \in C \). 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
\( \{ \text{EACH } o; \text{ IN } R \times S \ ( B ) \} \)
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 \( c; \) is a class and S is a set valued query expression, then
\( c; :- S \) and
\( c; :+ S \),
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

CONSTRANT B

is an integrity constraint definition.

Lock and unlock statements are defined as follows. If S is a set valued query expression, then
\( \text{READLOCK } S \) and
\( \text{WRITELOCK } S \)
are legal lock statements and
\( \text{UNLOCK } S \)
is a legal unlock statement in any transaction.