Desktop Schema Evolution
- Editing Schema

Takao MIURA
SANNO College
Kamikasuya 1573, Isehara, Kanagawa, JAPAN
E-Mail: miura@sanno.ac.jp

Abstract

In this paper we describe a general framework to do schema evolution. The main idea is desktop, the conceptual workspace to edit schemas. The evolution to be discussed is captured from several points of view; (1) Schema translation causes changing database states. (2) Three kinds of evolutions, type evolution, predicate evolution and complex type evolution are proposed. (3) Completeness properties are preserved. Also we discuss the implementation techniques.

1 Introduction

Recently new types of requirements in database processing capabilities have been increasing in several areas of applications, for example, personal computing. This is because the power of hardware has been surprisingly enhanced and because the proliferation of inexpensive personal computers and workstations has given to new classes of database application. We do have a special purpose database for a specific user; no textbook says this is a database. At the same time, a variety of sophisticated techniques have been developed to provide fast access mechanisms and powerful modeling capabilities. Mainly data independence means physical aspects such as converting indexing files into hashed files. One of the logical data independence is certainly to change database definitions as well as view definition. This is called schema evolution problem. However, less progress has been made in schema evolution of databases. Many traditional databases suffer from such difficulties.

Visual presentation of databases allows users to construct, maintain and query a database using a graphics-based interface and a consistent operation paradigm. The advantages are that they provide a high level of abstraction (modularity) to be understood easily, that they may display several parts of schemas simultaneously and that the modifications in possible by specifying parts of a diagram in any order while the operations could be validated interactively. Building and browsing database schemas, and constructing queries are accomplished using the same style of interface, that is, point and click interface to a database system.

The main goal of this paper is to establish the formalism on schema evolution and to examine the power using visual operations. Here we consider schema evolution as a map between database states. Schemas of databases are transformed by database users while the states of the schemas have to be automatically changed to keep correspondence. Putting it into other words, we discuss re-organization of databases. Not all the transitions are possible only by specifying formulas over schemas. Here in this paper we propose three kinds of schema evolutions, type evolution, predicate evolution and complex type evolution.

There have been a lot of researches in schema evolutions. It is possible to classify them into three types, that is, temporal, versioning and re-organizing. But our research focuses on the last point and let us restrict this point. In [9], a set of criteria have been proposed to reorganize relational databases using special operators. In ORION project [3, 4], a set of properties called invariants have been introduced in order to describe the essential behaviors of object-oriented paradigm such as scope inheritance and class lattices. The proposed operators have been proved complete. But the proof process assumes empty states of the type hierarchy; the database states are not always transformed automatically.

The evolution of E-R diagrams is discussed in [6]. They have investigated the properties of the relational schemas for E-R diagrams and shows the complete operators for the evolution.

Schemas virtualization [11] could be captured as a kind of schema evolution. By giving views they have defined virtual classes and discussed the ISA relationship. Note that this schema is a kind of views, the database states do not change by the evolution.

In section 2, we briefly describe a conceptual schema diagram based on a semantic data model AIS. Section 3 describes the type evolution and the operators. We prove that they can be characterized by boolean expression of sets. In section 4 we discuss the evolution of predicates and the completeness property. In section 5 we move to complex types. Section 6 contains our conclusion.

2 A PIM Model

PIM (Personal Information Management), an interactive graphics-based system is a general purpose schema management and query processing system for a semantic data model AIS (Associative Information Structure). A data model AIS supports the representation of entity types and complex object types, inter-type relationship, ISA relationship and constraints. Here we present a brief overview of AIS model based on several examples. A formal definition of the AIS model is presented in [1, 2].

2.1 A Data Model AIS

Information in AIS consists of entities, i.e., surrogates of objects in the real world, and associations, i.e., relationship among objects. An entity is said to have a type or an entity type if it shares certain common properties. One entity may have several types. An association is said to have a predicate if it has certain common properties. We denote an association among entities $e_1, \ldots, e_n$ with a predicate $p$ by $p(e_1, \ldots, e_n)$.
All associations have predicates which have types in each position according to the roles. The types involved are said defined types of the predicate denoted by \( p[E_1 \ldots E_n] \). All the defined types of a predicate are distinct. We note that there is no attribute concept found in other models, but AIS considers it as a special kind of relationship. Example 1(a) shows entities and entity types, (b) says a relationship \( \text{Take} \) between \( \text{Course} \) and \( \text{Prereq} \) types.

An association must be identified by a predicate name \( p \) and the defined types with the entities. We call this property a unique association assumption. Similarly an entity must be identified by its name and a type. However every entity must have a type \( \text{Entity} \) which means the name must be unique, we call this property a unique name assumption.

There is a natural graph representation of AIS schemas. Example 1(c) is a diagram which shows the information of example 1(a) and (b). As shown in the diagram, this is data level representation, an entity is shown using a small circle \( (\bullet) \). For example, \( \text{math}1 \) is shown with a label \( m1 \). A filled circle \( (\bullet) \) is used to indicate an association. A set of entities are connected together by this notation. For example \( [m2, cl] \) and \( [m2, c2] \) are the associations of \( \text{Take} \). Example 1(d) shows a diagram of schema level. Here an entity type is indicated by set notation and a predicate by a diamond \( (\bigcirc) \).

**Example 1** (See figure 1(c) and 1(d))

(a) Part of Entities and Types

<table>
<thead>
<tr>
<th>notation</th>
<th>entity types</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>math1, Course, Math</td>
</tr>
<tr>
<td>m2</td>
<td>math2, Course, Math</td>
</tr>
<tr>
<td>m3</td>
<td>math3, Course, Math</td>
</tr>
<tr>
<td>p1</td>
<td>phys1, Course, Physics</td>
</tr>
<tr>
<td>h1</td>
<td>chem1, Course, Chemistry</td>
</tr>
<tr>
<td>cl</td>
<td>col, Prereq</td>
</tr>
<tr>
<td>cl</td>
<td>cr1, Prereq</td>
</tr>
<tr>
<td>cl</td>
<td>cr2, Prereq</td>
</tr>
<tr>
<td>d1</td>
<td>desc1, Description</td>
</tr>
<tr>
<td>rl</td>
<td>score1, Score</td>
</tr>
<tr>
<td>el</td>
<td>emp1, Employee</td>
</tr>
<tr>
<td>sl</td>
<td>smith, Student, Employee</td>
</tr>
</tbody>
</table>

(b) Part of Associations

\[
\begin{align*}
\text{Take} & (\text{Course}(m2), \text{Prereq}(c1)) \\
\text{Take} & (\text{Course}(m2), \text{Prereq}(c2)) \\
\text{Take} & (\text{Course}(m3), \text{Prereq}(c3)) \\
\text{CourseInfo} & (\text{Course}(m2), \text{Description}(d1)) \\
\text{Register} & (\text{Course}(m2), \text{Student}(s1), \text{Score}(r1))
\end{align*}
\]

Above example describes a portion of a database; in this case there are three diamonds (labeled \( \text{Take, CourseInfo and Register} \)) and 8 sets, two of which are connected by the diamond. In the actual database, there may be many other data (or parts of schema) but the system may display parts of interest.

ISA relationship (or Generalization) is a kind of constraint of subtyping, represented by set inclusion in AIS. In example 1(d), inclusion relationship (for example, \( \text{Math} \subseteq \text{Course} \)) describes the constraints between the sets.

AIS provides a mechanism to describe Complex Objects by set and tuple constructors [5]. In AIS, complex objects are carefully captured by two aspects of abstraction, an entity and its constituents \( v(\text{value}) \). These two are explicitly combined to give special relationship called \( \text{isa} \), indicated by \( \lambda(e) = v \). Values of complex objects must obey structuring rules, complex type, and the explicit combination between types and complex types is said type structure. In Example 2, \( \text{Prereq} \) is explicitly connected to a set of \( \text{Course} \). This is indicated diagramatically by dotted lines; In the figures of Example 1(c) and (d), directed dotted lines show such combinations of data and types respectively. We note that this type structure does not contain all the possible values over \( \{\text{Course}\} \) but the subset. In other word, some of them are "filtered" through this connection.

Finally \( \Delta \) is defined to be a set;

\[
\Delta(P, E) = \bigcup_{m \geq 1} P \times E^n
\]

An element of \( \Delta(P, E) \) is denoted like \( p(e_1, \ldots, e_n) \) for \( p \in P, e_1, \ldots, e_n > \). \( \Delta(P, E) \) is denoted \( \Delta \) unless otherwise stated.

A database schema S is an ordered pair \( < T', P', \beta, \Psi, \Lambda > \) where \( T' \subseteq T, P' \subseteq P, \beta : P' \rightarrow 2^{T'}, \Psi \subseteq T' \times T' \) and \( \Lambda \subseteq T' \times \beta(T') \). A predicate \( p \) of degree \( n \) has \( n \) distinct primitive types \( E_1 \ldots E_n \), called defined types of \( p \), denoted by \( \beta(p) = (E_1 \ldots E_n) \) or by \( p[E_1 \ldots E_n] \). \( \Psi \) means a kind of constraints representing ISA. That is, \( t_1 \) is called a subtype of \( t_2 \) if \( (t_1, t_2) \in \Psi \). Informally we say that there is a (ISA) link from \( t_1 \) to \( t_2 \).

A database state of S is an ordered pair \( < E', A', \alpha, \lambda > \) where \( E' \subseteq \Delta, \alpha : E' \rightarrow 2^{T'} \) and \( \lambda : \Gamma_{E'}(t) \rightarrow \Gamma_{E'}(t') \) for each \( t, t' \in \Lambda \).

A database state is called valid if it satisfies the following:

\[
\Gamma_{E'}(t) = \{ e \in E' \mid t \in \alpha(e) \}
\]
functions r(t) + l?(Y). If there is no confusion, we define T'(t E t') the
range of the function x(l?(t)). Th e f unctions are not always surjective
and possibly subsets of r(t') are defined in the databases.

A'. We sometimes call this set <-set (of p).

Here we identify a set of operations for type evolution. When we talk
about P it is assumed that there are a set of types connected by (ISA)
links called type hierarchy. General schemas are discussed in the sub-
sequent sections and all the operations defined below are towards type
manipulation.

The operations are carefully de fined to have declarative property so
that the evolution proceeds by specifying portions of interests in paral-
lel. We show an evaluation algorithm that makes the evolution proce-
dures declarative. Also we discuss the completeness property.

3.1 Goals of Type Evolution

When we talk about type evolution in a naive manner, it is enough
to build five operations for type evolution[3]; add types, remove types,
rename type, add link and remove link. In fact, if all the types have
empty states, we can generate every set of types and links by these
operations. Moreover, we could give any database state using update
operation, i.e., state change within the same type. In this sense, we
already have a complete set of operations.

However the separation of type and state manipulation causes in-
creasing the number of operations needed, and their sequence must be
carefully specified to get the correct result. In example 1, assume we
want to replace Student by Working-Student. First we must define
a new type Working-Student by add-type, then we select working stu-
dents by Student∩Employee into the new type. Then we must rename
Student as Working-Student everywhere and finally remove the exten-
sion of Student. Each operation might be simple but the sequence of
the operations be exactly this one since 'remove Student' erases all the
information of the Students. Note that the completeness result does
not imply their integration.

The goals of the operations described below are now clear:

(1) Every operation causes state change.
(2) It is possible to specify portions of evolution.
(3) These operations have the completeness property under
some invariant conditions.
(4) It is natural to implement as an icon interface.

3.2 Editing Typecs

In order to define type evolution, we need some primitive concepts of
type expression, desktop and current type. Desktop is a conceptual en-
vironment for editing types. We can do create, destroy and change the
names and the states on this desktop while the actual type is unchanged
even if we are editing it.

A type t is a set of type structures t =< e1, . . . , en > and the database
D =< E, A', o, s >, type expression of a type t is < t, t', U > where t E
T, t' E T, U E E'.
Type expressions carry the transient information of types while editing.

Given a schema S =< T', P', p, @, A > and the database D =< E, A', o,
L >, type expression of a type t is < t, t', U > where t E T,
t' E T, U E E'.

In the subsequent, T' means T'.

3 Type Evolution

Here we identify a set of operations for type evolution. When we talk
about P it is assumed that there are a set of types connected by (ISA)
links called type hierarchy. General schemas are discussed in the sub-
sequent sections and all the operations defined below are towards type
manipulation.

The operations are carefully defined to have declarative property so
that the evolution proceeds by specifying portions of interests in paral-
lel. We show an evaluation algorithm that makes the evolution proce-
dures declarative. Also we discuss the completeness property.

3.1 Goals of Type Evolution

When we talk about type evolution in a naive manner, it is enough
to build five operations for type evolution[3]; add types, remove types,
rename type, add link and remove link. In fact, if all the types have
empty states, we can generate every set of types and links by these
operations. Moreover, we could give any database state using update
operation, i.e., state change within the same type. In this sense, we
already have a complete set of operations.

However the separation of type and state manipulation causes in-
creasing the number of operations needed, and their sequence must be
carefully specified to get the correct result. In example 1, assume we
want to replace Student by Working-Student. First we must define
a new type Working-Student by add-type, then we select working stu-
dents by Student∩Employee into the new type. Then we must rename
Student as Working-Student everywhere and finally remove the exten-
sion of Student. Each operation might be simple but the sequence of
the operations be exactly this one since 'remove Student' erases all the
information of the Students. Note that the completeness result does
not imply their integration.

The goals of the operations described below are now clear:

(1) Every operation causes state change.
(2) It is possible to specify portions of evolution.
(3) These operations have the completeness property under
some invariant conditions.
(4) It is natural to implement as an icon interface.

3.2 Editing Typecs

In order to define type evolution, we need some primitive concepts of
type expression, desktop and current type. Desktop is a conceptual en-
vironment for editing types. We can do create, destroy and change the
names and the states on this desktop while the actual type is unchanged
even if we are editing it.

A type t is a set of type structures t =< e1, . . . , en > and the database
D =< E, A', o, s >, type expression of a type t is < t, t', U > where t E
T, t' E T, U E E'.
Type expressions carry the transient information of types while editing.

Given a schema S =< T', P', p, @, A > and the database D =< E, A', o,
L >, type expression of a type t is < t, t', U > where t E T,
t' E T, U E E'.

In the subsequent, T' means T'.

3 Type Evolution

Here we identify a set of operations for type evolution. When we talk
about P it is assumed that there are a set of types connected by (ISA)
links called type hierarchy. General schemas are discussed in the sub-
sequent sections and all the operations defined below are towards type
manipulation.

The operations are carefully defined to have declarative property so
that the evolution proceeds by specifying portions of interests in paral-
lel. We show an evaluation algorithm that makes the evolution proce-
dures declarative. Also we discuss the completeness property.

3.1 Goals of Type Evolution

When we talk about type evolution in a naive manner, it is enough
to build five operations for type evolution[3]; add types, remove types,
rename type, add link and remove link. In fact, if all the types have
empty states, we can generate every set of types and links by these
operations. Moreover, we could give any database state using update
operation, i.e., state change within the same type. In this sense, we
already have a complete set of operations.

However the separation of type and state manipulation causes in-
creasing the number of operations needed, and their sequence must be
carefully specified to get the correct result. In example 1, assume we
want to replace Student by Working-Student. First we must define
a new type Working-Student by add-type, then we select working stu-
dents by Student∩Employee into the new type. Then we must rename
Student as Working-Student everywhere and finally remove the exten-
sion of Student. Each operation might be simple but the sequence of
the operations be exactly this one since 'remove Student' erases all the
information of the Students. Note that the completeness result does
not imply their integration.

The goals of the operations described below are now clear:

(1) Every operation causes state change.
(2) It is possible to specify portions of evolution.
(3) These operations have the completeness property under
some invariant conditions.
(4) It is natural to implement as an icon interface.
As a syntactic sugar, it seems convenient to have registered procedures. The procedure consists of a sequence of the operations with type parameters. It is perfectly legal to use procedures within a procedure even if they are recursively accessed. Procedures cannot be evaluated until the parameters are specified. This is one of the tricks to specify partial evolution of types. Here currency is denoted by dot (.)

Example 3 The following procedure is named WORKING(V) which returns working persons.

\[ \text{SELECT} (\text{V}) \]
\[ \text{INTERSECT} (\text{Employee}) \]

This procedure selects the parameter V then takes intersect with Employees of our database. For example, WORKING(Student) makes Working_Student.

There are two trivial assumptions to do the evaluation of the desktop; First, duplicate is not allowed at the evaluation among the (new) names of the type expressions \( \Omega_t \) of the desktop and the types not in the desktop. This is because the name of types in schema must be unique. Second, the types without 'name' are considered transient types to calculate state and they should be discarded.

Then the desktop is evaluated by the following rules. There are five possibilities to resolve the evaluation. Every type expression \(< t, t', U >\) can be classified and resolved as follows:

1. If both \( t \) and \( t' \) are \( \bot \), then this should be discarded.
2. If \( t = \bot \) and \( t' \neq \bot \), then add \( t' \) as a new type with \( \Gamma(t') = U \), that is, add \( t' \) to each \( \alpha(e) \) where \( e \in U \).
3. If \( t \neq \bot \) and \( t' = \bot \), then remove \( t \) from \( T' \) and remove \( t' \) from \( \alpha(e) \) where \( e \in \Gamma(t) \).
4. If \( t \neq \bot \) and \( t' \neq \bot \), then replace \( t \) by \( t' \), remove \( t \) from each \( \alpha(e) \) where \( e \in \Gamma(t) \) and add \( t' \) to each \( \alpha(e) \) where \( e \in \Gamma(t') \).
5. Types not on the desktop remain unchanged.

Example 4 Some of the types in an old database schema are:

\[ \text{Employee, Student, Teacher} \]

Assume current desktop over example1 contains:

\[ < \text{Student, WorkingStudent, } W >, < \text{Employee, Worker, } V > \]

Note there is no duplicate among the new names on the desktop and the names not on the desktop. The actions are taken according according to the rules:

1. \( \text{School} \) is added to the schema by the rule (2).
2. \( \text{Teacher} \) remains unchanged by (5).
3. \( \text{Employee} \) is replaced by \( \text{Worker} \) with \( V \) by (4).
4. \( \text{Student} \) is copied with a new name \( \text{WorkingStudent} \) and a new state \( U \) by (4), while the original \( \text{Student} \) is removed by (3).
5. The last expression is discarded by (1).

\[ \square \]

3.3 Completeness Property

As described previously, our main purpose of this investigation is to exploit a set of operators to get more semantics in a sense that type evolution causes correct state changes automatically. Completeness property shows us how rich the semantics carries.

The operators NAME and DUP (to generate \( t' \)) or DROP (to generate \( \bot \)) can be used to change the same \( t \). Clearly the state change has to be discussed.

Conceptually each of them has version number for the type. For example, the current type, say \( t \), merged with \( t \) keeps the same type and the same name. This means a new type \( t_{\bot} \) is generated by giving \( t_{\bot} = t \) if \( t \neq \bot \). In other words, it is possible to generate every boolean expression of types using our operators without calculating entity sets. To describe this concept formally, state expression is defined recursively as follows:

1. An empty set is a state expression.
2. For a primitive type \( t \), \( \Gamma(t) \) is a state expression.
3. For state expressions \( t_1, t_2 \), the intersection, the union and the difference of the two are also state expressions.

Theorem 1 (Type Completeness) A state of every type expression on the desktop is a state expression. Conversely every state expression can be generated as a state of some type expression on the desktop.

(Proof) Clearly every \( U \) of a type expression \(< t, t', U >\) on the desktop is a state expression because \( U \) should be generated by SELECT, NEW, DUP, DROP, UNION, INTERSECT, DIFFERENCE and LINK. Conversely, a state expression can be generated by the operations because union, intersection and difference are expressible by UNION, INTERSECT and DIFFERENCE respectively while empty sets by NEW, DROP and \( \bot \) sets by DUP.

Two constraints operators, LINK and DELINK, are for manipulating \( \Omega_t \) of the desktop. DELINK does not change the states at all but LINK does. Next theorem says that what activities we have to do at LINK.

Theorem 2 Assume that there are type expressions \( e_0 = < s, s', U_0 > \), \( e_1 = < t, t', U_1 > \) on the desktop and that \( (e_0, e_1) \) be added to \( \Omega_t \) by LINK operator. Let \( V = U_1 - U_0 \).

1. For each \( u, u', W > \) where we can reach from \( e_0 \) through ISA links in \( \Omega_t \), \( W \) is replaced by \( W \cup V \). Then, every constraint \( u, u' \) in \( \Omega_t \) is satisfied by the states, i.e., the state of \( u \) is contained in the state of \( u' \).
2. Conversely, for each \( e' = < u, u', W > \) where we can reach from \( e_1 \) through ISA links in \( \Omega_t \), if every constraint \( (e_0, e_1) \) in \( \Omega_t \) is satisfied by the new states, the new state of \( e_1 \) contains the old states of \( u \) and \( V \).
3. If we can get to \( e_1 \) from \( e_0 \) through ISA links, this constraint is already satisfied and the process needs not go any more.

(Proof) Assume \( e_0 < e_1 < \cdots < e_n \) in \( \Omega_t \). By the assumption the original states \( F_i, \cdots, F_n \) satisfy \( F_i \subseteq F_{i+1} \) for \( i = 1, \ldots, n \). After adding \( V \) to each \( F_i \), trivially the new states satisfy \( e_0 < e_1 < \cdots < e_n \) and we have (1).

Let \( E_i \subseteq E \) be the new states of \( e_0, e_1, \ldots, e_n \), and let \( c \) be the minimum number such that \( F_c \cap V \subseteq E_i \). Note that \( c \geq 1 \) by the definition, and that \( F_c \cap E_i \) because only union is allowed to the sets. Then there exists an element \( e \in F_c \cap V \setminus E_i \). Then \( e \) must be in \( V \), and therefore in \( E_i \). Since \( E_i \setminus E \), we have the contradiction for (2).

(3) is trivially by (1) and (2).

Let us comment on the complexity issue. The complexity of testing satisfiability of \( \Omega_t \) concerns the derivation of \( t' \) from \( t \) when \( (e_0, e_1) \) (i.e., \( n = 1 \)) are added where \( e_0 = < t, t', U_0 > \). Fortunately the linear time algorithm to obtain \( t' \) has been devised in [2] and all the tests can be done in polynomial time of the length of \( \Omega_t \) and the number of types in \( T' \).
3.4 An Evaluation Algorithm

When evaluating the desktop, there can be a naive evaluation; the states of type expressions be actually calculated whenever they are operated on the desktop. Clearly this approach is poor because successive manipulation might be time- and space-consuming.

Every state in type expressions can be expressed by state expression from empty sets and I-sets as described before. Then, instead of naive calculation, the sequence of operations could be stored as a state in type expressions. The sequence is, in fact, directed acyclic graph (DAG) called operation DAG whose leaf nodes are empty sets or I-sets, whose intermediate nodes are one of the labels of union, intersection and difference, and whose links show operation sequence.

Let us consider each operation. SELECT causes new leaf nodes whose states are F sets while NEW and DROP cause nodes of empty sets. UNION, INTERSECT, DIFFERENCE and LINK make intermediate nodes of the appropriate labels. DUP makes a copy of the operation DAG of the current type. NAME and DELINK do not give any effect.

The evaluation of the operation DAG is straightforward, i.e., bottom up calculation is needed to obtain the state of a node.

4 Predicate Evolution

Here we identify a set of operations for general schema evolution. Our operators are similar to type evolution but the semantics are separately described. The completeness property is discussed and the limitation is clarified.

4.1 Editing Predicates

The goals of the operations described below are just the same except the interaction of type evolution:

(1) Every operation causes state change.
(2) It is possible to specify portions of evolution.
(3) These operations have the completeness property under some invariant conditions.
(4) It is possible to exploit an iconic interface.
(5) Type evolution should be propagated to predicate evolution.

The primitive concepts such as predicate expression, desktop and current predicate are defined in a similar way.2

Given a database schema S =< T', P', β, μ, Ψ, Λ > and a database state D =< E', A', μ, Λ >, predicate expression of a predicate p is < p, p', N, F > where p E P - P', I means 'undefined' and 4 means an empty set. Currency is transformed into < p, N, F > on the desktop as < p, 1, N, 4 >. If the expression is current, the new currency becomes undefined.

DUP Current expression < p, p', N, F > is copied as a new expression < p, p', N, F >, where p1 is a new predicate. Currency is not changed. Optionally we can give a new name by DROP(n) so that < pr, p1, N, F > is generated.

INTERSECT(exp) The state of the currency < p, p', N, F > is taken with the state of the expression exp =< q, q', N', F' >; F is replaced by joining the two, F W F'. The other parts of this expression is unchanged. Note the defined names N must be same.

PROJECT(M) The set of defined types of the currency becomes A'.

5 Example

Assume there are two predicates in a schema, Take[Course, Prereq] and CourseInfo[Course, Description]. The former can be transformed into < Take, TakeDetail, (Course, Prereq, Description) U > by the following technique:

(1) Select Take[Course, Prereq] and CourseInfo[Course, Description] as expressions exp1, exp2 respectively.
(2) exp1 is OVERLAPPED with exp2. U is the join of the two states.
(3) The result is NAMEd by NAME(TakeDetail).

As in type evolution, there can be registered procedures. A procedure consists of a sequence of the operations with parameters of predicate expressions. Recursive procedures are allowed. Here, currency is defined by dot(.). Procedures cannot be evaluated until the parameters are specified.

Some kinds of deduction could be possible. For example, Ancestor relationship can be expressed by procedures.

Example 6 It is possible to define a procedure 'Ancestor' like below;
Ancestor

As this example shows, the fact that procedures can give the more expressive power although we don’t discuss any more[7].

There are two trivial assumptions to do the evaluation of the desktop,
(1) unique predicate name and (2) predicate expression without name be discarded. The evaluation rules of the desktop are almost same as type evolution and we skip the discussion.

4.2 Expressive Power

It is desirable to define the meaning of completeness by which predicate evolution causes correct state changes automatically. Several operators (NAME, DUP and DROP) can change the same name as type evolution. However as for states, the situation differs.

Looking at the operators, the reader will see relational algebra manipulation: union (UNION), intersection (INTERSECT), difference (DIFFERENCE), join (OVERLAP) and projection (PROJECT) over empty sets and ε-sets. There is no selection but this seems reasonable since state is never changed without changing schema. We call relational algebra without selection partial relational algebra. Then the next theorem holds.

Theorem 3 (Predicate Completeness) A state of every predicate expression on the desktop can be obtained by partial relational algebra. Conversely every partial relational algebra expression can be generated as a state of some expression on the desktop.

When evaluating the desktop, states of expressions can be actually described by the operational DAG whose internal nodes are partial relational algebraic operations, and let us skip the discussion.

5 Evolution of Complex Objects

In this section we discuss complex type evolution. Complex types are the structural properties of complex objects defined by 6 operations from T'. A set of operations for the evolution are exploited as an intepretation. Another difference is that we need new operators to split < Student, Score > into Student and Score.

On the other hand, if it is not desirable to separate type manipulation from state manipulation as we discussed in the previous sections. In this sense, the goals of the operations are just same as type and predicate evolutions.

Before developing our theory, let us add comments on two meanings of complex values. For two types t1, t2, the semantics of < t1, t2 > is defined to be the Cartesian product of the two sets of t1 and t2 whatever these two sets contain.

However it is possible to give a subset of the Cartesian product as an alternative semantics. For example, every prerequisite course takes several sets of courses but not all the possible sets. This fact means that a subset of the Cartesian product has to be given as the semantics. We call the latter position a value-based semantics of a complex type while the former a representation-based semantics. On the other hand, representation-based semantics can be calculated dynamically from the constituents and some sort of deduction is performed.

In our case, complex types are defined only in the context of type structures and it is enough to discuss only value-based semantics since another could be generated by type specification.

5.2 Editing Typcs

Desktop is extended to express complex types and values

\[ \text{T - NEW} \]

Create a new type structure; if the current type (described by NAME, DUP and DROP) can change the same name as type evolution. However as for states, the situation differs.

Looking at the operators, the reader will see relational algebra manipulation: union (UNION), intersection (INTERSECT), difference (DIFFERENCE), join (OVERLAP) and projection (PROJECT) over empty sets and ε-sets. There is no selection but this seems reasonable since state is never changed without changing schema. We call relational algebra without selection partial relational algebra. Then the next theorem holds.

Theorem 3 (Predicate Completeness) A state of every predicate expression on the desktop can be obtained by partial relational algebra. Conversely every partial relational algebra expression can be generated as a state of some expression on the desktop.

When evaluating the desktop, states of expressions can be actually described by the operational DAG whose internal nodes are partial relational algebraic operations, and let us skip the discussion.

In this section we discuss complex type evolution. Complex types are the structural properties of complex objects defined by 6 operations from T'. A set of operations for the evolution are exploited as an intepretation. Another difference is that we need new operators to split < Student, Score > into Student and Score.

On the other hand, if it is not desirable to separate type manipulation from state manipulation as we discussed in the previous sections. In this sense, the goals of the operations are just same as type and predicate evolutions.
Example 7 Two complex types \(< \text{Student}, \text{Score} > \) and \(< \{ \text{Student} \} \)
are not compatible so that UNION is not allowed. On the other hand, \(< \text{Student}, \text{Score} > \lor \{ \text{Employee} \} \) is compatible with \(< \text{Student}, \text{Employee} > \lor \text{Score} \).\(\Box\)

There are another class of operators by which complex types are generated. Note the above class cannot generate new complex types. Complex types can be represented by DAGs (tree) whose leaf nodes are primitive types and whose internal nodes are labeled with one of set, \(\lor\), and \(\land\). For a node \(p\) in the tree, a subtree of the tree rooted at \(p\) is called \(p\) part of the tree, while a part of the complex value that matches \(p\) part is said \(p\) part of the value. Then the evolution set, or, \(\land\), and tuple. For a node \(p\) in the tree, a subtree of the tree rooted at \(p\) is called \(p\) part of the tree, while a part of the complex value that matches \(p\) part is said \(p\) part of the value. Then the evolution set, or, \(\land\), and tuple.

Our operators are as follows where \(p\) points a node in the current tree and \(\exp\) a complex type tree (if \(p\) is omitted, 'root' is assumed).

\[
\text{CUP}(\exp, p) \quad \text{Assume } \exp = \langle t', co, co', U \rangle \text{ and the currency } \langle t, co, co', V \rangle \text{ and } p \text{ points } c \text{ in the current. Intuitively, } co = (\prod (c \cdot \ldots) \cdot \ldots) \text{ is replaced by } (\prod (c \lor co') \cdot \ldots) \text{ where } \ldots \text{ of each } e' \in U \text{ is in the result for an element } \ldots (e) \cdot \ldots \in V.
\]

Formally, a new complex type \(< t, co, co', V' \rangle \) is generated as follows; The node \(c\) of \(co'\) is replaced by or node whose left child points \(p\) part and the right points \(co'\) resulting in \(co''\). Initially \(V'\) is set to \(V\). Then, for each element in \(V\), all the elements whose \(p\) part is an elements of \(U\), are also added into \(V'\).

\[
\text{CAP}(\exp, p) \quad \text{Assume } \exp = \langle t', co, co', U \rangle \text{ and the currency } \langle t, co, co', V \rangle \text{ and } p \text{ points } c \text{ in the current. Intuitively, } co = (\prod (c \cdot \ldots) \cdot \ldots) \text{ is replaced by } (\prod (c \land co') \cdot \ldots) \text{ where } \ldots \text{ of each } e' \in U \text{ is in the result for an element } \ldots (e) \cdot \ldots \in V.
\]

Formally, a new complex type \(< t, co, co', V' \rangle \) is generated as follows; The node \(c\) is replaced by and node whose left child points \(p\) part and the right points \(co'\) resulting in \(co''\). Only elements of \(V\) whose \(p\) part are also in \(U\), are added into \(V'\).

\[
\text{TUPLE}(\exp, p) \quad \text{Assume } \exp = \langle t', co, co', U \rangle \text{ and the currency } \langle t, co, co', V \rangle \text{ and } p \text{ points } c \text{ in the current. Intuitively, } co = (\prod (c \cdot \ldots) \cdot \ldots) \text{ is replaced by } (\prod (c \land co') \cdot \ldots) \text{ where } \ldots \text{ of each } e' \in U \text{ are added to the result for an element } \ldots (e) \cdot \ldots \in V.
\]

Formally, a new complex type \(< t, co, co', V' \rangle \) is generated as follows; The node \(c\) is replaced by tuple node whose left child points \(p\) part and the right points \(co'\) resulting in \(co''\). For each element in \(V\), all the elements whose \(p\) part is replaced by a pair \(< e, e' >\) where \(e'\) is an element of \(U\), are added into \(V'\).

\[
\text{SET}(p) \quad \text{Assume the currency is } \langle t, co, co', V \rangle \text{ and } p \text{ points } c \text{ in the current. Intuitively, } co = (\prod (c \cdot \ldots) \cdot \ldots) \text{ is replaced by } (\prod (\{e\} \cdot \ldots) \cdot \ldots) \text{ where an element } \ldots (\{e\}) \cdot \ldots \text{ is added to the result for an element } \ldots (e) \cdot \ldots \in V.
\]

Formally, a new complex type \(< t, co, co', V' \rangle \) is generated as follows; The node \(c\) is replaced by set node whose only one child points \(p\) part resulting in \(co''\). For each element in \(V\), an element whose \(p\) part value \(c\) is replaced by a set \(\{e\}\) is added into \(V'\).

\[
\text{SPLIT}(p) \quad \text{Assume } \exp \text{ points a node } c \text{ in the current tree. For the currency } \langle t, co, co', V \rangle, \text{ one or two new complex type } \langle t, co, V' \rangle \text{ and } \langle t_2, \land, co', V'' \rangle \text{ are generated where } t_2 \text{ is a new type;}
\]

Example 8 \(\text{CUP and SET are exemplified.}\)

(1) Let \(< t_1, t_2 > \) be the complex type of the current with the state below.
After applying \(\text{CUP}(\langle t', \{ts\}, U \rangle, t_2)\), we have the following result;
\[
\begin{array}{c|c|c|c}
\text{t}_1 & \text{t}_2 & \{\text{t}_3\} & \text{t}_1 \lor \text{t}_2 \lor \{\text{t}_3\} \\
1 & 2 & \{10\} & 1 \lor 2 \\
1 & 3 & \{11,13\} & 1 \lor \{10\} \\
2 & 2 & \{11,13\} & 1 \lor 3 \\
2 & 2 & \{10\} & 1 \lor 3 \\
2 & 2 & \{11,13\} & 1 \lor 3
\end{array}
\]

(2) Let \(t_1 \lor t_2 \) be the complex type of the current with the state below. After applying \(\text{SET}(\langle t', t_2, U \rangle, t_2)\), we have the following result; \(\Gamma(t_1) = \{1,2\}, \Gamma(t_2) = \{2,3\}\)
\[
\begin{array}{c|c}
\text{t}_1 \lor \text{t}_2 & \text{t}_1 \lor \{\text{t}_2\} \\
1 & 1 \\
2 & 2 \\
\{2\} & 2 \lor 3
\end{array}
\]

Note that two values 2 and \{2\} appear since 2 are of types \(t_1\) and \(t_2\).

\(\Box\)

The evaluation rules of the desktop of complex types are similar to the previous cases and we omit them.

5.3 Completeness Property

The names can be assigned any time we like. Clearly the state change has to be discussed. In type evolution, state expressions provide a boolean class consisting of an empty set, \(\Gamma\)-sets structured by \(U, t\) and the difference. In this case of complex type evolution, the possible states from a given set of complex values through re-structuring operators are described by extended state expressions.
Before defining them, some new kinds of data structure operators are introduced. Let $U, U_1, U_2$ be sets of complex values of $t, t_1, t_2$ be complex types, i.e. $U \subseteq \Gamma(t)$, $U_1 \subseteq \Gamma(t_1)$ and $U_2 \subseteq \Gamma(t_2)$. Also let $p$ point a node of $t, q$ a node of $t$.

$p$-Union $U_1 \cup p U_2$ is a set $U_1 \cup \{(e') \mid (e') \in U_1 \text{ and } e' \in U_2\}$ where $e$ means the $p$ part.

$p$-Intersect $U_1 \cap p U_2$ is a set $\{(e) \mid (e) \in U_1 \text{ and } e \in U_2\}$ where $e$ means the $p$ part.

$p$-Difference $U_1 \rightarrow p U_2$ is a set $\{(e) \mid (e) \in U_1 \text{ and } e \not\in U_2\}$ where $e$ means the $p$ part.

$p$-Tuple $U \times p U_2$ is a set $\{(e, e') \mid (e, e') \in U_1 \text{ and } e' \in U_2\}$ where $e$ means the $p$ part.

$q$-Project $\Pi_q(U)$ is a set $\{(e) \mid (e) \in U_1 \}$ where $\langle e, e' \rangle$ means the $q$ part. $\Pi_q(U)$ is similarly defined.

$q$-Typing $\Gamma_q(t)$ is a set $\{(e) \mid (e) \in U_1 \text{ and } e \in \Gamma(t)\}$ where $e$ means the $q$ part.

Now (extended) state expressions are defined recursively as follows:

1. An empty set is a state expression.
2. For a primitive type $t$, $\Gamma(t)$ is a state expression.
3. For a type structure $\langle t, \omega \rangle$, $U = \Gamma(\langle t, \omega \rangle)$ is a state expression.
4. For state expressions $U_1, U_2, U_3, U_4$, $U_1 \cup U_2, U_2 \cap U_3, U_1 \rightarrow U_2, U_4 \times p U_2$ representing the union, the intersection and the difference of the two at a 'position' $p$ in $U_1$ are also state expressions.
5. For a state expression $U$ and a complex type, $\Pi_q(U)$ and $\Gamma_q(t)$ are state expressions.

Theorem 4 (Complex Type Completeness) A state of every type expression on the desktop is a state expression. Conversely every state expression can be generated as a state of some type expression on the desktop.

(Proof) The story is similar to the type completeness. $\square$

6 Conclusion

In this paper, three kinds of schema evolutions, type evolution, predicate evolution and complex type evolution have been proposed. For each class of evolution, we have identified a set of operators and shown that they are complete, i.e., they can generate every schema with the desired class of states. Also we have discussed the evaluation procedures.

In a companion paper [8], we have proposed a visual language called PIM Algebra that is suitable for personal database environment. This language and the operators for the evolution would provide other new kinds of database application. The design and the implementation of this system is now undergoing.

References