TR-87-31

# Meta-Data Management

by

Leo Mark
Nick Roussopoulos

# META-DATA MANAGEMENT

by
Leo Mark
Nick Roussopoulos

## Abstract

A database system should support a rich variety of **meta-data** describing and controlling the management and use of data. Few present database systems provide even rudimentary integrated meta-data management facilities.

In this paper we briefly describe the architecture of a **Self-Describing Database System**, which has an **active and integrated data dictionary system** providing the only source of meta-data to users, programs, and database system, and using the services offered by the database system for meta-data management.

We use the relational data model to design part of a **self-describing meta-schema** and the **update dependency** formalism to specify some of operation controlling the evolution of database schemata.

We explain how to use the meta-schema and the operation specifications to set up and expand a **data dictionary schema**, which describes and controls the management and use of the database system.

Special attention is given to the problem of specifying and controlling the **inter-level propagation** of schema changes to data changes.

## 1. Architecture for Self-Describing Database Systems

A Self-Describing Database System and its environment is illustrated in figure 1, [ 1, 2].

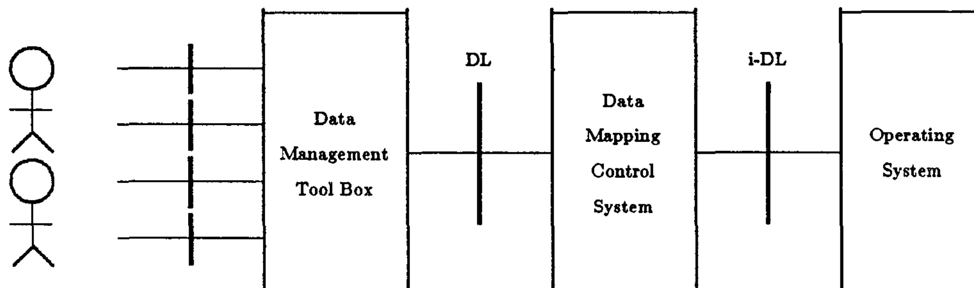



Figure 1. Self-Describing Database System Environment

**The Data Mapping Control System**, DMCS, supports and enforces two orthogonal dimensions of data description, the **point-of-view dimension** and the **intension-extension dimension**.

Research supported in part by the Systems Research Center, University of Maryland under the National Science Foundation Grant OIR-85-00108.

**The point-of-view dimension has three levels of data description: the information meaning described in the conceptual schema**, the external data representations described in **external schemata**, and the internal physical data structure layout described in the **internal schema**. These three levels of data description result in databases which are flexible and adaptable to changes in the way users view the data and in the way data is stored. This flexibility and adaptability is usually called data-independence, [3].

The intension-extension dimension has four levels of data description: the information about the data model supported by the database system described in the **meta-schema**, the information about the management and use of data described in the **data dictionary schema**, the information about specific applications described in **application schemata**, and the **application data**. Each level of data description in the intension-extension dimension is the intension of the succeeding data description and the extension of the preceeding data description. An intension completely describes and controls changes of an extension. All the levels of the intension-extension dimension are described in terms of the same data model. A description of the meta-schema is explicitly stored as part of its own extension - it is **self-describing!** The stored meta schema can be retrieved, but it can not be changed; it is self-describing, not self-destructing.

The DMCS can be thought of as a DBMS which, on one hand is stripped to the bones, but on the other hand is still a complete DBMS. It supports the set of elementary functions that are essential to the maintenance of the two dimensions of data description.

**The Data Language interface**, DL, is the data manipulation language for the data model. Because the meta-schema is self-describing all data, including descriptions, is defined, retrieved, and manipulated through the DL interface; no data definition language is needed.

The **Data Management Tool Box** contains software which is plug-compatible with the DMCS through the DL-interface. A data management tool supports high level functions which are important, but not essential to data management. To produce a plug-compatible data management tool it is important that information about the data model can be retrieved through the DL-interface, and this is exactly why the meta-schema is explicitly stored. Each of these tools will have its own user interface, but it must interface with the DMCS through the DL-interface. A database administrator would develop or buy off-the-shelf data management tools, such as schema design aids, software documentation packages, high level query language processors, report generators, etc., to supplement the elementary functions supported by the DMCS. A database administrator would develop or buy off-the-shelf definitions of the data dictionary schema and definitions of application schemata for a class of applications.

**The Internal Data Language interface**, i-DL, is the interface through which all data is passed from the DMCS to the Operating System supporting the DMCS.

The two orthogonal dimensions of data description, the point-of-view and the intension-extension dimension, supported by the DMCS are illustrated in figure 2.

Any system in this new architecture is born with data structures to hold the meta-schema extension - the data dictionary schema. Initially, the data dictionary schema consists of the stored description of the meta-schema. The data dictionary schema can subsequently be expanded into a full data dictionary schema describing and controlling the management and use of application databases. Initially, the extension of the data dictionary schema - the data dictionary data - consists of an empty set of tables.

The architecture supports **software plug-compatibility** and **data plug-compatibility** which gives the database administrator the freedom to choose the tools for data management and define the data management strategy he finds best suited for his enterprise.

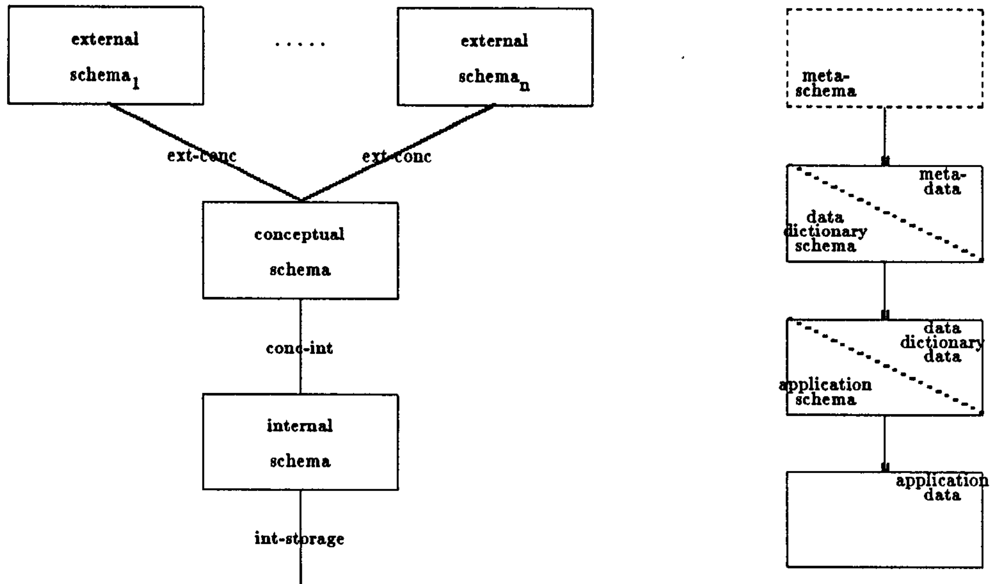



Figure 2. Point-of-View Dimension and Intension-Extension Dimension

The architecture unifies well-established principles and current trends in the database and the data dictionary system areas. It is a generalization of the ANSI/SPARC DBMS Framework which only considers the point-of-view dimension; it comprises recent ideas on the intension-extension dimension from the ISO WG on the Conceptual Schema and Information Base; it supplements data model standardization efforts; and it supplements data dictionary system standardization efforts.

The architecture is based on the notion of **Self-Describing Database Systems** and has matured through several discussions in the ANSI/SPARC Database Architecture Framework Task Group, DAFTG. The architecture has been accepted by ANSI/SPARC and is being considered by ISO as a reference model for database systems in the late 1980's and the 1990's.

## 2. The Meta-Schema

In this paper, we shall use the relational data model as an example, [4].

We shall use a graphic formalism inspired by the object-role data models, [ 5, 6]. The symbols in figure 3 represent the definition of a relation with name 'r' and two attributes with names 'a1' and 'a2'. The attributes are defined over a non-lexical domain with name 'd1' and a lexical domain with name 'd2', respectively, [ 7, 8].

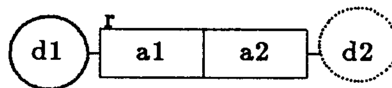



Figure 3. Graphic Formalism for Data Structures

The entities in relational schemata are definitions of relations, domains, and attributes. The entity-names used in these definitions are relation-names, domain-names, and attribute-names. Accordingly, we define in figure 4 the non-lexical domains named 'relation', 'domain', and 'attribute' and the lexical domains named 'relation-name', 'domain-name', and 'attribute-name'. The three unary relations named 'relation', 'domain', and 'attribute' define sets of

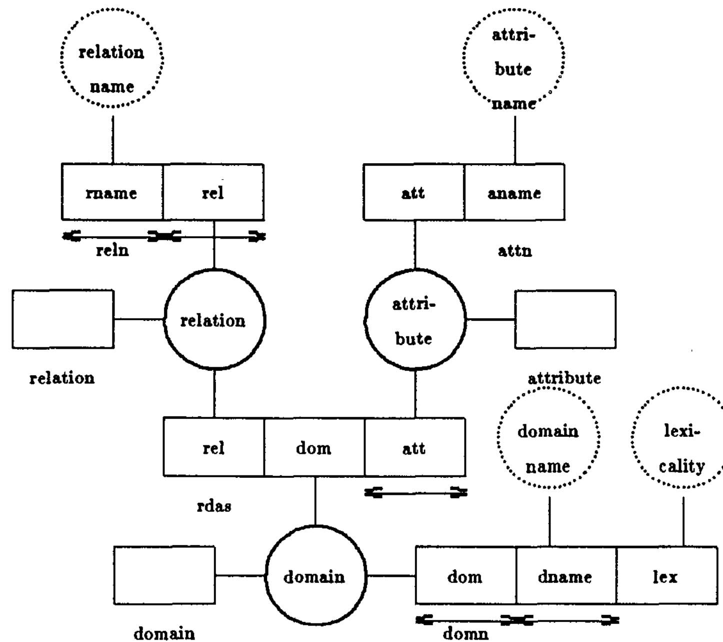
entities in existence. The relation named 'reln' define relationships between existing relations and their relation-name. The relation named 'attn' defined relationships between attributes and attribute-names. The relation named 'domn' define relationships between domains and their domain-names and lexicality. Finally, the relation named 'rdas' define the relationships between relations, domains and attributes. The keys, indicated by double headed arrows, point out attributes, the values of which uniquely identify tuples in the relation. As can be seen, no two relations can have the same name, and no relation has more than one name. The same restriction applies to domains and their names. In 'rdas' we indicate that an attribute is a unique entity related to at most one domain and relation.

There are several more constraints involved: the relations 'relation', 'domain', and 'attribute' model sets of entities in existence. Therefore, reln[relation] $\subseteq$ relation[relation], and corresponding rules applies for domains and attributes. On the other hand, we will insist that all relations do have names, so actually reln[relations] = relation[relation]. Also, the rule about unique attribute names within relations has not yet been modeled.

These rules, together with the keys indicated above, and several other rules, will all be specified as part of the operations to be defined in the meta-schema.

Boxes represent meta-schema relations. Full circles represent non-lexical domains of surrogates used to model entities. Broken circles represent lexical domains used to model entity-names. Arrows represent keys. The meta-schema is so far "self-describing"; it is defined in terms relations, domains and attributes, and its definition can be stored in the database it defines. See figure 5. (We have omitted the unary relations: 'relation', 'domain', and 'attribute')

Figure 4. Core Meta-Schema



reln

| rname: relation name | rel: relation |
|---|---|
| reln | r1 |
| domn | r2 |
| attn | r3 |
| rdas | r4 |

domn

| dom: domain | dname: domain name | lex: lexicality |
|---|---|---|
| d1 | relation | non-lexical |
| d2 | attribute | non-lexical |
| d3 | domain | non-lexical |
| d4 | relation name | lexical |
| d5 | attribute name | lexical |
| d6 | domain name | lexical |
| d7 | lexicality | lexical |

rdas

| reln: relation | dom: domain | att: attribute |
|---|---|---|
| r1 | d4 | a1 |
| r1 | d1 | a2 |
| r2 | d3 | a8 |
| r2 | d6 | a9 |
| r2 | d7 | a10 |
| r3 | d2 | a3 |
| r3 | d5 | a4 |
| r4 | d1 | a5 |
| r4 | d3 | a6 |
| r4 | d2 | a7 |

attn

| att: attribute | aname: attribute name |
|---|---|
| a1 | rname |
| a2 | rel |
| a3 | att |
| a4 | aname |
| a5 | rel |
| a6 | dom |
| a7 | att |
| a8 | dom |
| a9 | dname |
| a10 | lex |

Figure 5. Meta-Schema Description Stored in Meta-Schema Extension

## 3. Operations in the Core Meta-Schema

To make sure that the meta-schema completely models and controls all operations on its extension - the data dictionary schema - we define an insert, delete, and modify operation for each relation in the meta-schema. From the database administrator's point-of-view these operations are elementary, but as we shall see in their specification below, each involves several implied operations on update dependent relations. All the operations we define must be explicitly represented in the meta-schema, otherwise the meta-schema does not completely model and control all operations on its extension. For the meta-schema to be self-describing, it is necessary that it explicitly models the notion of an operation and controls all operations on the specification of operations. We shall not consider this expansion of the core meta-schema in this paper. For a detailed description, see [9]. Operations are specified in the following language, [10]:

### 3.1 Syntax

Each operation is defined by a set of update dependencies each of which has the following form:

<op1> --> <cond>,<op2>, ...,<opn>.

where <op1> is the operation being defined, <opi>, $2 < i < n$, is either an implied operation or an implied primitive operation, and <cond> is a condition on the database state.

An operation <opi> has one of the following forms:

- insert(<relation_name>(<tuple_spec>))
- delete(<relation_name>(<tuple_spec>))
- modify(<relation_name>(<tuple_spec>),
  <relation_name>(<tuple_spec>))

where the <tuple spec> is a tuple variable for the relation with the name <relation name> and consists of a list of <domain variable>s. The <tuple spec> in <op1> is the formal parameter for <op1>. All the <domain variable>s in the <tuple spec> of <op1> are assumed to be universally quantified. All <domain variable>s in the <tuple spec>s of <opi>, that are not bound to a universally quantified <domain variable> in <op1>, are assumed to be existentially quantified. All <domain variable>s are in caps; nothing else is.

The implied primitive operations are: 'assert' for adding a new tuple in a relation, 'retract' for eliminating one, 'write' and 'read' for retrieving data from the user, 'new' for creating a unique new surrogate, and 'break' for temporarily stopping the system to do some retrievel before giving the control back to the system. The implied primitive operations <opi> have the following forms:

- assert(<relation name>(<tuple spec>))
- retract(<relation name>(<tuple spec>))
- write('<any text>'), or write(<domain variable>)
- read(<domain variable>)
- new(<relation name>(<tuple spec>))
- break

The <relation name> used in the operation 'new' must be the name of a unary relation defined over a non-lexical domain. The conditions <cond> are expressions of predicates with the form

<relation name>(<tuple spec>). The connectives used in forming the expressions are '∧' (and) and '¬' (negation). In addition, we use the primitive predicates 'nonvar' and 'var' to decide whether or not a <domain variable> has been instantiated.

Conditions can also be used by the user to retrieve data from the system.

## 3.2 Semantics

An operation succeeds if, for at least one of its update dependencies, the condition evaluates to true and all the implied operations succeed. It fails otherwise.

When a operation is invoked its formal parameters are bound to the actual parameters. The scope of a variable is one update dependency. Existentially quantified variables are bound to values selected by the database system or to values supplied by the interacting user on request from the database system. Evaluation of conditions, replacement of implied operations, and execution of implied primitive operations is left-to-right and depth-first for each invoked update dependency. For the evaluation of conditions we assume a closed world interpretation [11].

The non-deterministic choice of a replacement for an implied operation is done by backtracking, selecting in order of appearance the update dependencies with matching left-hand sides. If no match is found, the operation fails.

An implied operation matches the left-hand side of an update dependency if:

- the operation names are the same, and
- the relation names are the same, and
- all the domain components match. Domain components match if they are the same constant or if one or both of them is a variable. If a variable matches a constant it is instantiated to that value. If two variables match they share value.

The semantics of the primitive operations are:

- assert(r(t)); its effect is r := r ∪ {t}; it always succeeds; all components of 't' are constants.
- retract(r(t)); its effect is r := r\{t} where all components of 't' are constants. It always succeeds.
- write('text'); it writes the 'text' on the user's screen. It always succeeds.
- write(X); writes the value of 'X' on the user's screen. It always succeeds.
- read(X); reads the value supplied by the user and binds it to 'X'. It always succeeds (if the user answers).
- new(r(D)); produces a new unique surrogate, [7], from the non-lexical domain over which 'r' is defined and binds the value of the variable 'D' to this surrogate. It always succeeds.
- break; suspends the current execution and makes a new copy of the interpreter available to the user, who can use it to retrieve the information he needs to answer a question from an operation.

The list of primitive operations is minimal for illustrating the concept. It can easily be extended. It is emphasized that primitive operations are not available to the user; he cannot directly invoke them.

The execution of 'assert' and 'retract' operations done by the system in an attempt to make an operation succeed, will be undone in reverse order during backtracking. This implies, that an operation that fails will leave the database unchanged.

## 3.3 The Operations

We shall only specify the few operations in the table below to illustrate the principles. For a detailed discussion, see [9].

| rel/op | insert | delete | modify |
|---|---|---|---|
| domain | | | |
| domn | | | |
| relation | x | x | |
| reln | x | x | |
| attribute | x | x | |
| attn | | | |
| rdas | x | x | |

```
insert(relation(R))
 →  var(R),
    new(relation(R)),
    insert(relation(R)).
 →  nonvar(R) ∧ relation(R).
 →  nonvar(R) ∧ ¬(relation(R)),
    assert(relation(R)),
    insert(rdas(R,_,_)),
    insert(reln(_,R)).
```

If the variable 'R' is uninstantiated when the insertion into relation 'relation' is called, then the system produces a new surrogate and proceeds with the insertion. If 'R' is instantiated there are two possibilities: 'R' is already in relation 'relation' in which case the insertion succeeds with the database state unchanged; or 'R' is not in relation 'relation' in which case we insert it, and since all relations must have some attributes and a name we propagate by triggering insertions into the relations 'rdas' and 'reln' of the tuples (R,_,_) and (_,R), respectively, indicating that the relation surrogate is at this point the only thing we know.

```
insert(reln(N,R))
 →  var(R),
    new(relation(R)),
    insert(reln(N,R)).
 →  var(N),
    write('relation name?'),
    break,
    read(N),
    insert(reln(N,R)).
 →  nonvar(N) ∧ nonvar(R) ∧ reln(N,R).
 →  nonvar(N) ∧ nonvar(R) ∧ ¬(reln(N,_)) ∧ ¬(reln(_,R)),
    assert(reln(N,R)),
    insert(relation(R)).
```

The two first rules produce or request from the user any uninstantiated variable values. The insertion operation succeeds with the database state unchanged if a relation with that particular name is already in the database. If no relation with the name 'N' exists and the relation represented by surrogate 'R' does not already have a name we assert the tuple and propagate by inserting 'R' in relation 'relation'.

```
insert(attribute(A))
 →   var(A)),
     new(attribute(A)),
     insert(attribute(A)).
 →   nonvar(A) ∧ attribute(A).
 →   nonvar(A) ∧ ¬(attribute(A)),
     assert(attribute(A)),
     insert(attn(A,_)),
     insert(rdas(_,_,A).
```

The first update dependency produces a new attribute surrogate if needed. The second succeeds if the surrogate is already present. The third makes the assertion and propagates through insertions in 'attn' and 'rdas'.

```
insert(rdas(R,D,A))
 →   nonvar(A) ∧ rdas(_,_,A).
 →   var(A),
     new(attribute(A)),
     insert(rdas(R,D,A)).
 →   var(R) ∧ ¬(nonvar(A) ∧ rdas(_,_,A)),
     write('relation surrogate?'),
     break, read(R),
     insert(rdas(R,D,A)).
 →   var(D) ∧ ¬(nonvar(A) ∧ rdas(_,_,A)),
     write('domain surrogate?'),
     break, read(D),
     insert(rdas(R,D,A)).
 →   nonvar(A) ∧ nonvar(R) ∧ nonvar(D) ∧ ¬(rdas(_,_,A)) ∧
    ¬(rdas(R,_,B) ∧ attn(A,N) ∧ attn(B,N) ∧ ¬(A=B)),
     assert(rdas(R,D,A)),
     insert(relation(R)),
     insert(domain(D)),
     insert(attribute(A)).
```

If needed, a new attribute surrogate is produced. If the attribute is not already in 'rdas' and relation- and domain-surrogates are not provided, then they are requested. Finally, if attribute name uniqueness within relations is not about to be violated, then the tuple is asserted in 'rdas'. This may cause propagation to 'relation' and 'domains' which is taken care of; and, finally, propagation to 'attribute' is called.

```
delete(rdas(R,D,A))
 →  ¬(rdas(R,D,A)).
 →  var(A) ∧ var(D) ∧ var(R),
    write('nothing done').
 →  nonvar(A) ∧ rdas(R,D,A) ∧ rdas(R,_,B) ∧ ¬(A=B),
    retract(rdas(R,D,A)),
    delete(attribute(A)).
 →  nonvar(A) ∧ rdas(R,D,A) ∧ ¬(rdas(R,_,B) ∧ ¬(A=B)),
    retract(rdas(R,D,A)),
    delete(attribute(A)),
    delete(relation(R)).
 →  nonvar(D) ∧ var(A) ∧ var(R) ∧ rdas(R,D,A),
    delete(rdas(R,D,A)),
    delete(rdas(_,D,_)).
 →  nonvar(R) ∧ var(A) ∧ var(D) ∧ rdas(R,D,A),
    delete(rdas(R,D,A)),
    delete(rdas(R,_,_)).
 →  nonvar(R) ∧ nonvar(D) ∧ var(A) ∧ rdas(R,D,A),
    delete(rdas(R,D,A)),
    delete(rdas(R,D,_)).
```

The relation 'rdas' is central in the core meta-schema. Since we want domains to be allowed to exist without being currently used in any relations, the only direct propagation from deleting tuples from 'rdas' is to 'attribute', and sometimes to 'relation'.

The definition of the compound update operation above is very special, because it allows the user free use of any combination of unistantiated variables possibly resulting in multiple tuple deletions. This is obtained through extensive use of recursion, and it automatically propagates every single tuple update during the process.

If no tuples matching the actual parameter is present in 'rdas' the operation succeeds with the database being unchanged. If all variables are uninstantiated we don't allow any change. The operation succeeds with nothing done to the database. There are now the following possible combination of instantiated/uninstantiated parameter combinations left:

| rdas | ( R, | D, | A) |
|---|---|---|---|
| | - | - | a |
| | r | - | a |
| | - | d | a |
| | r | d | a |
| | - | d | - |
| | r | - | - |
| | r | d | - |

Because the attribute uniquely identify one tuple, one (domain, relation), all operations with an attribute surrogate cause one tuple deletion from 'rdas', which is propagated to 'attribute'. If the attribute to be deleted is the last one in a relation, then the deletion also propagates to 'relation'. If only the domain-surrogate is given, rdas(_,d,_), then we must delete all uses of that domain in any relation. If only the relation-surrogate is given, rdas(r,_,_), then all attributes for that relation is deleted from 'rdas', implying of course the deletion of the relation too.

Finally, if both relation and domain-surrogates are given, rdas(r,d,_), then we must delete all uses of the given domain in the given relation.

```
delete(attribute(A))
 →   ¬(attribute(A)).
 →   nonvar(A) ∧ attribute(A),
     retract(attribute(A)),
     delete(rdas(_,_,A)),
     delete(attn(A,_)).
 →   var(A),
     write('attribute surrogate?'),
     break,
     read(A),
     delete(attribute(A)).
```

If the attribute is not present in 'attribute' the operation succeeds with the database unchanged. In the last update dependency, if no value is given, the user is prompted for one and the operation is tried again. In the second update dependency, if a value is given and that value is in 'attribute' we remove it and propagate by deleting all its names from 'attn' and all its uses from 'rdas'.

It takes only simple arguments to see, that if a deletion of an attribute is caused by a previous deletion from 'rdas', then the propagation to 'rdas' from this update dependency immediately returns with success, because that attribute is not used in 'rdas' anymore (by the very first update dependency).

If, on the other hand, this deletion from 'attribute' is the original one, the propagation back to 'attribute' later following the propagation back to 'rdas', will succeed by the first update dependency above.

```
delete(relation(R))
 →   ¬(relation(R)).
 →   var(R),
     write('relation surrogate?'),
     break,
     read(R),
     delete(relation(R)).
 →   nonvar(R) ∧ relation(R),
     retract(relation(R)),
     delete(reln(_,R)),
     delete(rdas(R,_,_)).
```

If a surrogate for the relation is given, and the relation exists, it is removed, its name is deleted, and all attribute and domain relationships to the relation is deleted from 'rdas'.

```
delete(reln(N,A))
 →   ¬(reln(N,R)).
 →   var(N) ∧ var(R),
     write('relation name?'),
     break,
     read(N),
     delete(reln(N,_)).
 →   ¬(var(N) ∧ var(R)) ∧ reln(N,R),
     retract(reln(N,R)),
     delete(relation(R)).
```

Only the relation surrogate or the name is needed to uniquely identify a tuple to be deleted, so there is no internal recursion in this rule. The propagation of the deletion to 'relation' succeeds with the job done, and the returning call of a deletion from 'reln' stops on the first rule.

In a fully expanded meta-schema there will be several more relations and operations modeling and controlling all operations on the specification of operations [9].

# 4. Farther Down<br>The Intension-Extension Dimension

In the previous section we defined some of the relations and operations of a meta-schema that describes and enforces general rules and laws for schema definition in terms of relations and operations. We shall now concentrate on some of the important aspects farther down the intension-extension dimension. A more detailed discussion of these aspects can be found in [9].

Obviously, we need to control schema definition from both the meta-schema level and the data dictionary schema level. The first question is therefore what the initial content of the intension-extension dimension is. And, the second question is how we set up the initial content of the intension-extension dimension.

As we shall see, the initial content of the data dictionary schema only includes the bare minimum needed to control the definition and change of application schemata. A data dictionary schema must do much more than that. The third question therefore is how the database administrator chooses, defines, and enforces a database management strategy.

The relations and operations in the meta-schema only define and control operations on the immediate extension of the meta-schema. They only define and control intra-level propagation of a schema change. Insofar that general rules and laws can be identified, which is hard, the meta-schema and the data dictionary schema should also define and control inter-level propagation caused by changing an extension which is interpreted as intension for the next level. The fourth question therefore is how we specify and enforce inter-level propagation.

These four questions will be addressed in the following short sections.

## 4.1 Initial Content of the Intension-Extension Dimension

We need to control the definition and change of relations and operations from both the meta-schema and the data dictionary schema. In the first case, it is the definition and change of the data dictionary schema we must control, and in the second case, it is the definition and change

of the application schemata we must control.

The relations and operations defined in the previous section are not **level specific** and we can use them at any level where we need to control the definition and change of schemata at the next level.

We therefore choose to include the relations and operations defined in the previous chapter in both the meta-schema and the data dictionary schema; this is also in accordance with our principle of self-description.

On the other hand, we do need to be level specific when we call and execute an operation on some relation.

We shall choose to identify the level of data description a level specific operation is defined at by prefixing operation names with an 'm' for meta-schema, a 'dd' for data dictionary schema, and an 's' for application schema.

A set of level specific meta-schema operations can be defined in the following way:

```
m_delete(reln(N,R))
→   c1,
    delete(reln(N,R)).
```

```
m_insert(reln(N,R))
→   c2,
    insert(reln(N,R)).
    ..
    ..
    ..
```

The purpose of the conditions c1,c2, .... is to protect from changes that part of the data dictionary schema, which has the twofold role of being a stored description of the meta-schema and being an integral part of the data dictionary schema controlling application schema definition and change.

As a simple example of how the 'm' operations protect the stored description of the meta-schema from modifications, consider the condition 'c1' in the operation specification above. It must include the following:

$$c1 = \neg(N{=}name_1) \wedge \neg(N{=}name_2) \wedge \ldots \wedge \neg(N{=}name_n),$$

where name_1, ... , name_n are the names of the meta-schema relations.

We choose to let the initial content of the data dictionary data be an empty set of relations ready to hold the extension of the initial content of the data dictionary schema.

We finally choose not to include any initial content of application data.

Our choices for the initial content of the intension-extension dimension can be summarized as follows:

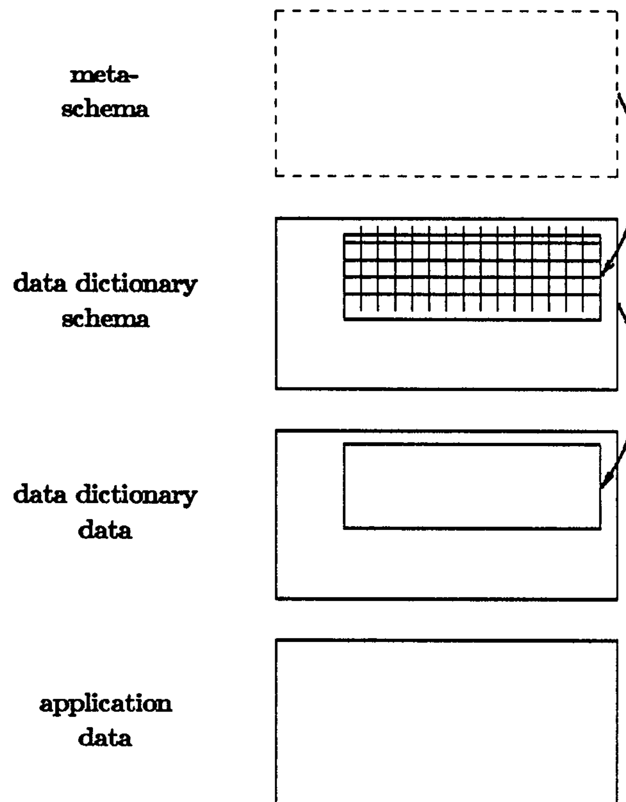



Figure 6. Initial Content of the Intension-Extension Dimension

The meta schema consists of the meta-schema relations and the operations defined in the previous section and the level specific meta-schema operations defined above. The meta-schema itself is imaginary, meaning that nothing is actually stored in the dotted line box. We shall later explain how the relation definitions of the meta-schema itself are hard-wired in the DL-processor.

A description of the meta-schema is explicitly stored in the black box of the data dictionary schema. Nothing in the black box can be changed by operations defined in the meta-schema - the meta-schema is self-describing, not self-destructing. It is the conditions in the level specific meta-schema operations that prevent changes of the stored description of the meta-schema. Nothing, but the description of the meta-schema, is part of the initial content of the data dictionary schema.

The initial content of the data dictionary data is an empty set of relation tables ready to store the extension of the initial data dictionary schema.

## 4.2 How to Set Up the Initial Intension-Extension Dimension

In order to explain how we set up the initial content of the intension-extension dimension we must first make some assumptions about the DL-processor.

**The DL-processor:**

The DL-processor - the DMCS - has the definition of the meta-schema built in. This basically means, that it knows the names of the meta-schema relations, how they are structured, and on which files their extensions are stored. The DL-processor does not have the definition of the meta-schema operations built in. Instead it has a **search module** which retrieves an operation specification given an operation call. It locates 'm' operations from the data dictionary schema, 'dd' operations from the data dictionary schema, and and 's' operations from the data dictionary data. The **execution module** executes non level specific operations relative to the level specific operations which called them; and if the level can not be decided from the call, then the operation fails. The DL-processor enforces the operational semantics of the update dependency formalism. As part of this, the specification of primitive operations is built in.

**The Set-Up:**

To set up the initial content of the intension-extension dimension, we replace the **search module** of the DL-processor by a **booster module.** The booster module searches an externally stored set of level specific meta-schema operation specifications to retrieve an operation specification given an operation call. This **externally stored** set of level specific meta-schema operations is different from the set we want to store in one sense, the operations do **not** have the conditions $c_1, c_2, \dots, c_n$ defined above.

We now issue the series of 'm' operation calls, resulting in the insertion of the meta-schema description as part of the data dictionary schema.

Finally, we replace the booster module by the search module, and we are in business.

Note, that an insertion into the relation 'relation' at any level creates an empty tabel at the next level to hold the extension of the inserted relation definition. This is a a general rule for inter-level propagation, which will be discussed later in this section.

## 4.3 How to Expand the Data Dictionary Schema.

The data dictionary schema must define and control all operations on data used for database management. Most importantly, the data dictionary schema must control the definition and change of application schemata as discussed above. But, in addition to that, the data dictionary schema must define and control the notions of authorization, user, schema, program, file, distribution, etc.

As the database administrator decides on a database management strategy he will define relations and operations in the data dictionary schema to enforce this strategy, and the part of the data dictionary schema copied from the meta-schema will gradually be expanded into a full data dictionary schema. This means, that the level specific operations he defines in the data dictionary schema will be more complicated, than those for the meta-schema. As the data dictionary schema is expanded, the level specific operations in the data dictionary schema must control the propagation of changes of application schemata to changes of data dictionary data controlled by the expanded data dictionary schema.

In an expanded data dictionary schema the level specific operations are constructed from the original meta-schema operations as illustrated below.

```
dd_delete(reln(N,R))
→   c1,
    delete(reln(N,R)),
    ..
    ..
→   c2,
    delete(reln(N,R)),
    ..
    ..
→   cn,
    delete(reln(N,R)),
    ..
    ..
```

where the conditions, c1, c2, ... , cn, can test data controlled by the expansion of the data dictionary schema, and thereby distinguish alternatives for propagating changes of data, controlled by the initial data dictionary schema, into changes of data, controlled by the full data dictionary schema. The alternative sequences of implied operations on the data, controlled by the expanded data dictionary schema, are inserted in the operation specification above, as indicated by the dots.

We note, that database administrator must use the original non level specific insert, delete, and modify operations, when he defines the level specific data dictionary schema operations, that constitutes the user interface for the database designer.

It is really important to realize, that the only part of the data dictionary schema, which cannot be changed in any way through the meta-schema, is the initial part controlling relation definition and change.

This means, that the database administrator can design the database management application to suit the particular needs of his enterprise, the same way a database designer designs an ordinary application. If a particular kind of authorization procedure is preferred by the database administrator, then he can include it. If the database needs to be distributed, then he can define the distribution model he wants. In summary, the database administrator decides on the **database management strategy** himself, rather than having to suffer with an inadequate strategy forced upon him by the database system vendor. Or, since the meta-schema is explicitly stored, several independent software houses may offer off-the-shelf plug-compatible database management strategies - that is, data dictionary schema definitions - which the database administrator can choose from. Such plug-compatible data dictionary schema definitions could only be produced by the original vendor if the meta-schema was not explicitly described.

Our database system framework supports the notions of **plug-compatible data** and **plug-compatible software.** Therefore, we can strip the DMCS to the bones, leaving only the DMCS functions, which are absolutely essential. We know of no other database system or data dictionary system born this naked; they all come with more of the database management strategy built in, e.g. an authorization strategy, and with built in Data Management Tools which are nice and important, but not essential.

In summary, the simplicity and potential of our framework is based on

- the explicitly stored meta-schema description which gives the plug-compatibility;
- the explicitly stored and changable data dictionary schema allows us to design our own database management strategy; and
- the power of the update dependency formalism which allows us to fully follow the 100% principle, which means that an intension completely controls its extension and thereby relieves the DL-processor from enforcing a lot of special rules.

We end this subsection with a simple example, to illustrate how the database administrator may expand the data dictionary schema.

**Example**

The database administrator can define a simple authorization strategy for all application database users by defining the following relation and operations in the data dictionary schema using the 'm'-operations:

authorized

| user_name | relation_name | operation_name |
|---|---|---|

```
dd_insert(authortized(U,R,O))
→    ..
     ..
     assert(authorized(U,R,O)).
```

```
dd_delete(authorized(U,R,O))
→    ..
     ..
     retract(authorized(U,R,O)).
```

When the database designer defines an application schema relation and some operations on it, they will look as follows:

supplier

| s# | s_name | city |
|---|---|---|

```
s_insert(supplier(S,N,C))
→    authorized(U,supplier,s_insert),
     assert(supplier(S,N,C)).
```

```
s_delete(supplier(S,N,C))
→    authorized(U,supplier,s_delete),
     retract(supplier(S,N,C)).
```

(We assume, that the DL-processor knows the username U.)

When the database designer has defined the operations above, using the 'dd'-operations, he inserts tuples in the relation 'authorized', allowing the users, he wants, to do the operations, he wants.

When a user 'u1' calls, e.g. s_insert(supplier(..)), the operation will succeed only if the tuple (u1,supplier,s_insert) is stored in the relation 'authorized'.

This was the simple case, where the database administrator trusts, that the database designer will remember to insert the condition on the relation 'authorized', in all update dependencies of all 's'-operations, he defines. If the database administrator wants to make sure of this, he must define the 'dd'-operations on the relation 'condition' to force all conditions of 's'-operations to include "authorized(U,<relation_name>,<operation_name>)". When the database designer later calls the 'dd'-operations to define 's'-operations, there is no way he can exclude or forget the condition on the relation 'authorized' in any of the update dependencies constituting the 's'-operations.

## 4.4 Inter-Level Propagation

In the previous chapter we gave a specification of operations defining and controlling schema updates and their intra-level propagation.

We shall now study the inter-level propagation of a schema update, that is, the effect a schema update has on the data defined and controlled by the schema.

**Example**

**DD-schema:**

reln

| rname | rel |
|---|---|

```
dd_delete(reln(N,R))
→   ..
    ..
```

**Schema:**

person

| p# | family_name |
|---|---|

```
s_delete(person(P,M))
→   ..
    ..
```

**Data:**

person

| | |
|---|---|
| p1 | oconnor |
| p2 | stamenas |
| . | .. |
| . | .. |

If we want to delete the tuple (person,r1) from 'reln', then we make the following operation call: dd_delete(reln(person,r1)). This operation will only affect tuples in the "database" defining the schema. We would somehow like all the tuples in the relation 'person' to be deleted too.

There are three ways in which we can try to handle the problem of inter-level propagation and they are all needed. We can

- include a set of rules for inter-level propagation in the semantic definition of the DL;
- specify inter-level propagation directly in the operation specifications; or
- provide a data management tool for database reorganization.

**Including Inter-Level Propagation in the Semantics of the DL.**

Only general rules for inter-level propagation for our data model should be included in the semantic definition of the DL.

Before we define these rules, let us illustrate one of the pitfalls of the problem of inter-level propagation by considering the deletion of an attribute from a relation with a non empty extension. If we think, that the general rule for inter-level propagation in the relational model in this situation is enforced by projecting the extension of the relation over the remaining attributes, then we are in for a surprise. The problem is not, that we don't know which duplicate tuples, if any, to get rid of. The problem is, that the relational projection operator has nothing to do with the process of deleting an attribute from a relation definition. We may in some situations decide, that when we delete an attribute from a relation definition, then the extension of the new relation should be computed by projection, but this is not a general rule for the relational model. The reason for our surprise probably is, that we have gotten so used to the production of some artificial intension of a relation every time we use the relational operators on some relation extensions. What we must realize is, that **the relational operators have no intensional semantics,** that is, the intension produced by the relational operators has no meaning; it must be assigned by humans [12].

We know of only very few general rules for inter-level propagation in the relational data model:

(1) A relation definition can be deleted, if the extension of the relation is currently empty. This rule is implemented in [13] and [14].

(2) A relation definition can be inserted; the extension of the relation will defined to be empty. This rule is implemented in [13] and [14].

(3) An attribute definition can be deleted from a relation definition, if the extension of the relation is currently empty. This rule is implemented in [13] and [14].

(4) An attribute definition can be inserted in a relation definition, if the extension of the relation is currently empty. This rule is implemented in [14]. The rule is generalized in [13], where an attribute definition can be inserted in a relation with a non empty extension. The corresponding values in the extension of the relation are defined to be null, representing "value unknown" or "value inapplicable."

(5) A domain definition can be deleted, if it is not part of any relation definition. A domain definition can be deleted from a set of relation definitions, if all the attributes defined over the domain can be deleted from the relation definitions. The inter-level propagation is via the deletion of the attribute definitions.

(6) A domain definition can be inserted in a relation definition, if the implied attribute definition can be inserted in the relation definition. The inter-level propagation is via the insertion of the attribute definition.

(7) A view definition can be deleted without any inter-level propagation. This rule is implemented in [13].

(8) A view definition can be inserted without any inter-level propagation. This rule is implemented in [13].

We shall include the general rules (1) - (8) in the semantics of the DL. This means, that when we insert a base relation definition the system will create an empty table to hold the extension of the relation, and when we delete a base relation definition the system will remove the empty table.

All of the above rules for inter-level propagation, except for the generalization of rule (4), actually specify an inter-level propagation which is nil; except for the empty tabels which are set up or removed.

To help the database administrator and the database designer bring the database into a state which allows a schema update to take place, we can let the non level specific operations in the meta-schema give a couple of hints, as illustrated in the example below.

**Example**

In order to enforce a generalization of rule 1, to allow the deletion of a relation with a non empty extension, we could change the definition of the delete operation on relation 'reln' as follows:

```
delete(reln(N,R))
→   reln(N,R) ∧ op_spec(O,R) ∧ opn(O,M) ∧ ¬(¬(M=dd_delete) ∧ ¬(M=s_delete)),
    write('Delete all tuples in'),
    write(N),
    write('using operation'),
    write(M),
    break,
    retract(reln(N,R)),
    delete(relation(R)).
→   ..
    ..
    ..
```

The condition in the operation checks that there exists a delete operation for the level in question. The relations 'op_spec' and 'opn' are two of the relations in the expanded meta-schema controlling the definition and change of operation specifications. The operation simply tells the user which inter-level propagation he must take care of before passing the control back to the operation.

General rules for inter-level propagation are more interesting in a data model which support the notion of "sub-type" or "is-a" relationships. A set of rules is defined in [15].

**Specifying Inter-Level Propagation in the Operations.**

Data dependent rules for inter-level propagation can be explicitly modeled in the operations. If the rules are general for the data model, then they should be included in the non-level specific operations in the meta-schema. If the rules apply to a specific application of the data model, then they should be included in the level specific operations.

### Example

Suppose we define two relations, "person_name(p#, name)" and "person_address(p#, address)", with compound update operations enforcing a referential integrity constraint from 'person_address' to 'person_name'. This means, that when we insert the tuple (pi, address) in 'person_address', then a tuple (pi, name) must be present in or inserted into 'person_name', and vice versa for deletion. Suppose we want to generalize this rule to the relation definitions themselves, meaning that if we delete the relation, 'person_name', then we want to delete the definition of the relation, 'person_address' too.

We can specify this rule in the data dictionary schema as follows:

```
dd_delete(reln(N,R))
→   ..
    ..
    ..
→   reln(N,R) ∧ op_spec(O,R) ∧ opn(O,s_delete) ∧ (N=person_name),
    s_delete(person_name(_,_)),
    retract(reln(person_name,R)),
    dd_delete(reln(person_address,P)),
    dd_delete(relation(R)),
→   ..
    ..
    ..
```

The condition of the operation checks that there exists an operation with name 's_delete' for the relation to be deleted. The relations 'op_spec' and 'opn' are two of the relations in the expanded meta-schema used to model and control the definition of operations. The operation simply enforces the data dependent rule, that if the relation definition to be deleted is for the relation, 'person_name', then all tuples in the extension of this relation must first be deleted.

This technique works very well on data dependent rules for inter-level propagation. The technique can be used both in the level specific operations in the meta-schema and the data dictionary schema.

### Database Reorganization

When there is nothing else we can do, the user must resort to tools for database reorganization. This means, that the database administrator or the database designer will be responsible for the inter-level consistency of of intensions and extensions.

Database reorganization is often a very elaborate process involving a system shutdown. Only few systems, including SystemR [13] and QBE [14], support on-line database reorganization. A very powerful algebra for database reorganization was proposed for the extended relational model RM/T [16].

On-line database reorganization is supported by a Self-Describing Database System. The general technique is the following:

(1) Insert the definition of the new set of relations.

(2) Insert the definition of the compound update operations for the new relations.

(3) Write a Data Management Tool, that uses the delete operations on the old relations and the insert operations on the new relations to move the data, and call it.

(4) Delete the definitions of the old relations.

(5) Rename the new relations.

This completes our discussion of the Intension-Extension Dimension.

## Summary.

We have briefly described the architecture of a Self-Describing Database System, which has an active and integrated data dictionary system providing the only source of meta-data to users, programs, and database system, and using the services offered by the database system for meta-data management. This architecture has been accepted by the ANSI/SPARC and is being considered by the ISO as a reference model for database systems in the late 1980s and 1990s, and we will undoubtedly see several database systems develop in this direction.

We then described the core of the meta-schema in the intension-extension dimension using the relational data model as an example. We introduced a language based on the concept of update dependencies between relations to describe the operations on a database, and described some of the operations through the core meta-schema using this language.

In the last section, we used the meta-schema to set up the initial contents of the intension-extension dimension. We explained how to expand the data dictionary schema. And, we discussed the important aspect of inter-level propagation of schema modifications in the intension-extension dimension.

# REFERENCES


(1) Mark, L. and Roussopoulos, N.: "The New Database Architecture Framework - A Progress Report." In Proceedings IFIP WG 8.1 Working Conference on Theoretical and Formal Aspects of Information Systems, Sitges, Spain, 1985.

(2) Burns, T., Fong, E., Jefferson, D., Knox, R., Mark, L., Reedy, C., Reich, L., Roussopoulos, N., and Truszkowski, W.: "Reference Model for DBMS Standardization." Report from the Database Architecture Framework Task Group of the ANSI/X3/SPARC Database System Study Group, Sigmod Records, March 1986.

(3) Tsichritzis, D. and Klug, A. (eds.): "The ANSI/X3/SPARC DBMS Framework." Information Systems, Vol. 3, No. 3, 1978.

(4) Codd, E.F.,: "A Relational Model of Data for Large Shared Data Banks." Communications ACM 13, No. 6, 1970.

(5) Breutman, B., Falkenberg, E., and Mauer, R., "CSL, A Language for Defining Conceptual Schemas." In Bracchi, G. and Nijssen, G.M. (eds.), Database Architecture, North Holland, 1979.

(6) Nijssen, G.M.: "One, Two or Three Conceptual Schemata". In Bracchi, G. and Nijssen, G.M. (eds.), Database Architecture, North Holland, 1979.

(7) Hall, P., Owlett, J., and Todd, S.: "Relations and Entities." In Nijssen, G.M. (ed.), Modelling in Data Base Management Systems, North Holland, 1976.

(8) Griethuysen, J.J. van (ed.): "Concepts and Terminology for the Conceptual Schema and Information Base." ISO/TC97/SC5/WG3 - N695, 1982.

(9) Mark, L.: "Self-Describing Database Systems - Formalization and Realization," TR-1484, Department of Computer Science, University of Maryland, 1985. (Ph.D. Dissertation).

(10) Mark, L. and Roussopoulos, N.: "Update Dependencies," submitted to the 1985 VLDB Conference.

(11) Reiter, R., "On Closed-World Data Bases." In Gallaire, H. and Minker, J. (eds.), Logic and Data Bases, Plenum Press, 1978.

(12) Roussopoulos, N.: "Intensional Semantics of A Self-Documenting Relational Model," TR-1264, Department of Computer Science, University of Maryland, College Park, Maryland 20742, 1982.

(13) Chamberlin, D.D. et al.: "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control." IBM Journal of Research and Development 20, No. 6, 1976.

(14) Zloff, M.M.: "Query-by-Example: A Data Base Language." IBM Systems Journal 16, No. 4, 1977.

(15) Roussopoulos, N. and Mark, L.: "Schema Manipulation in Self-Describing and Self-Documenting Data Models." Accepted for publication, International Journal of Computer and Information Science.

(16) Codd, E.F.: "Extending the Database Relational Model to Capture More Meaning." ACM Transactions on Database Systems 4, No. 4, 1979.