TR-87-37

Operational Specification of Update Dependencies

by

Leo Mark

Nick Roussopoulos

TR-87-37

Operational Specification of Update Dependencies

by

Leo Mark

Nick Roussopoulos

Operational Specification of Update Dependencies

Leo Mark & Nick Roussopoulos

Department of Computer Science
and
Systems Research Center
University of Maryland
College Park, Maryland 20742
U.S.A.

Contents

Operational Specification of Update Dependencies

Leo Mark & Nick Roussopoulos
Department of Computer Science
and
Systems Research Center
University of Maryland
College Park, Maryland 20742
U.S.A.

Abstract

Update dependencies are the basis of a new formalism for **operational database specification.**

Update dependencies can be used to specify aspects of databases ranging over **integrity constraints, transactions, normalization, view maintenance and update, and metadata management.**

In many situations, the update dependencies can be **automatically** produced as an implication of, e.g. defining and normalizing a database schema.

The set of update dependencies specifying a database constitute a **production system.** Update dependencies can be **executed** and tested, and form a concise, precise, and **implementation independent** basis for implementation.

Future research issues include an **algebra** for update dependencies and a set of **control structure abstractions.** The algebra would allow complementary update dependencies to be automatically combined. The control structure abstractions would give a more user friendly specification language.

1. Update Dependencies - Introduction

We introduce update dependencies as a means to specify and control the semantics of a database under update. A set of update dependencies give an operational declarative specification of an update on a relation in terms of a set of alternative sequences of implied updates on the relation and possibly on other relations, and specifies the conditions under which the implied updates must succeed for the original one to succeed.

Update dependencies can be used to specify aspects of databases ranging over **integrity constraints, transactions, normalization, view maintenance and update, metadata management.**

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

A major difference between this approach and conventional approaches to integrity constraint specification and enforcement is, that rather than specifying a set of rules for what are the valid states of the database, we specify a set of rules for how the database can evolve from one valid state to another. As an implication of a schema definition or change, we can automatically generate update dependencies that specify and enforce a wide variety of conventional integrity constraints, including keys, primary keys, referential integrity, functional and multi-valued dependencies, total function constraints, cardinality constraints, numerical constraints, subset constraints, set exclusion constraints, set partition constraints, is-a constraints, etc. Another difference is, that a system controlled by update dependencies does not bluntly reject an update of a relation when a constraint is about to be violated, it tries to execute a set of implied updates that in many situations lead to the desired consistent state.

The notion of a database **transaction** [4] can be used to group together a set of database updates, and to suspend the checking of integrity constraints until all the updates have been carried out. To make this work, all the integrity rules should be explicitly represented in the database and the database system should be able to check that the database state is consistent with the integrity constraints. Since only few database systems supports any integrity constraints at all, transactions are usually just used to group a set of update which the application programmer "knows" must be executed together. A step in the right direction is offered by the event-procedure concept [1]. An event-procedure checks an integrity rule triggered by a database transaction. If the transaction violates an integrity rule, the event-procedure executes a violation-action [5] which may have the nature of a corrective-action. This notion of corrective-action is a very useful concept and one can think of update dependencies as being a generalization of this concept. A system based on update dependencies is flexible and cooperative - it takes over some of the work involved in forming acceptable transactions. The system receives an update against the database and makes all the additional updates needed to arrive at a new consistent database state. Sometimes the system may not be able to complete the transaction on its own in which case it will ask the user for the additional information. However, the information on what data is required to carry out the update and from where it would be obtained (the database or the user) is recorded in update dependencies. Furthermore, we use a declarative representation of update dependencies which is very flexible to change.

In spitę of the well-known consistency problems, update anomalies, etc. many real-life databases are not and will not be normalized. Reasonable objections to **normalization** are that it splits up information that is naturally used together in an application, it makes queries more complicated, and it decreases performance. Although the definition of views alleviate the first two problems, it has an adverse effect on performance; furthermore the views do not support updates. Update dependencies can solve the problem of keeping redundant representations of a fact consistent during update by simply changing all the representations of the fact whenever one of the representations is changed. In fact, update dependencies can be automatically generated to enforce the non-full and the transitive functional dependencies and maintain the consistency in an un-normalized database. Another problem with normalization is that we can only get to 3.NF if we want to preserve the functional dependencies we started out with; going from 3.NF to BCNF may cause loss of functional dependencies. Again, we can automatically generate update dependencies that maintain the inter-relational functional dependencies that are lost by going to BCNF.

Only a restricted set of the views on a database are theoretically updatable, but in several situations we know precisely what we would like the effect of an update of a particular view to be. The problem with views is that in the process of defining them, we sometimes throw away some of the information the system needs in order to update them. Update dependencies allow us to add this information again. Update dependencies allow us to define policies for updating views. Sometimes it pays of to incrementally update a view rather that recomputing it when the base relations it is derived from are updated. Update dependencies allow us to give a precise specification of the algorithm used.

Correct metadata management is essential in a database. There is no point in worrying about consistency and integrity at the data level if things are messed up at the metadata level. Update dependencies can be used to specify and enforce constraints at the metadata level as well as the data level, as discussed above. But, in addition to this, update dependencies can be used to specify and enforce the implications on data from changing the metadata. That is, if we know a rule that we would like to have automatically enforced on the data whenever we change the metadata, then we can use the update dependencies to specify and enforce this rule. Among the possible rules of this nature are rules for database reorganization, rules for deleting non-empty relations or attributes, rules for adding attributes to non-empty relations, and rules for deleting relations as an implication of the deletion of other relations.

Finally, update dependencies can be used for a variety of watch-dog tasks, such as collecting database usage statistics, security checking, logging, etc.

Related work on transaction specification and the goal oriented approach to constraint satisfaction, that has influenced this work, can be found in the TAXIS system [2,6,8,10,11]. The concept of defining higher abstractions on existing models such as the relational systems is in line with that of the direction taken in [12,13,16]. The notion of a control abstraction has been more fully exploited for software specification in [15]. The use of these abstractions to maintain semantic integrity constraints has been influenced by the constraint connections of the Structural Model [14].

In section 2 we present the syntax and semantics of the specification language for update dependencies. In section 3 we discuss some of the applications of update dependencies, including integrity constraint, transactions, normalization, views, and metadata management. In section 4 we briefly describe the implementation of an interpreter for the update dependency specification language. Section 5 finally contains our conclusions and describes issues for future research.

2. The Specification Language

A relational database consists of a set of relations that must obey a set of intra- and inter-relational integrity rules. Because of the inter-relational integrity rules only few database updates are atomic, that is, confined to updates within one relation. We introduce update dependencies as a means to specify and control non-atomic transactions on relational database.

To model this the database schema must contain declarative operational definitions of three compound update operations:

- insert
- delete
- modify

for each relation defined in the schema. The only way a user can change the database state is by invoking one of the compound update operations. This means, that the new operations are primitive to the user, but compound from the point of view of the system; just like the operations of abstract datatypes. Other application dependent operations can be defined as well, but before we do that we shall concentrate on the general ones above, from which they are built.

We shall first define the syntax and the semantics of update dependencies. We then discuss application dependent update operations and give some guidelines for how to program with update dependencies.

2.1 Syntax

Each compound update operation is defined by an update dependency with the following form:

$$
\begin{array}{ll}
\multicolumn{2}{l}{<op>} \\
\rightarrow & <c_1>, \\
 & <op_{1,1}>, \\
 & <op_{1,2}>, \\
 & .. \\
 & <op_{1,n1}>. \\
\rightarrow & <c_2>, \\
 & <op_{2,1}>, \\
 & <op_{2,2}>, \\
 & .. \\
 & <op_{2,n2}>. \\
\rightarrow .. &
\end{array}
$$

where <op> is the compound update operation being defined, $<op_{ij}>$, is either an implied compound update operation or an implied primitive operation, and $<c_i>$ is a condition on the database state.

A compound update operation $<op_i>$ 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 <op> is the formal parameter for <op>. All the <domain variable>s in the <tuple spec> of <op> are assumed to be universally quantified. All <domain variable>s in the <tuple spec>s of $<op_{ij}>$, that are not bound to a universally quantified <domain variable> in <op>, are assumed to be existentially quantified. All <domain variable>s are in caps; nothing else is.

The implied primitive operators are: 'add' for adding a new tuple in a relation, 'remove' 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 $<op_{ij}>$ have the following forms:

- add(<relation name>(<tuple spec>))
- remove(<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. The connectives used in forming the expressions are '∧' (and) and '¬' (negation). The predicates are of the form <relation name>(<tuple spec>) to determine whether or not a given tupel is in a given relation; or of the form 'nonvar(X)' or 'var(X) to decide whether or not a <domain variable>, X, has been instantiated; or of the form X<comp>Y, where <comp> is a comparison operator.

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

2.2 Semantics

A compound update operation succeeds if, for at least one of the alternatives in the its update dependency, the condition evaluates to true and all the implied operations succeed. It fails otherwise.

When a compound update 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 compound update 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 [4].

The non-deterministic choice of a replacement for an implied compound update 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 compound update 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:

- add(r(t)); its effect is r := r $\cup$ {t}; it always succeeds; all components of 't' are constants.
- remove(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, [5], 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 'add' and 'remove' operations done by the system in an attempt to make a compound update operation succeed, will be undone in reverse order during backtracking. This implies, that a (user invoked) compound update operation that fails will leave the database unchanged.

2.3 Application-Oriented Update Dependencies

We now relax the syntax of an update dependency to allow the definition of application-oriented compound update operations. An update dependency has the form:

```
<op>
 →  <c_1>,
    <op_1,1>,
```

$<op_{1,2}>$,
..
$<op_{1,n1}>$.
→ $<c_2>$,
$<op_{2,1}>$,
$<op_{2,2}>$,
..
$<op_{2,n2}>$.
→..

where <op> is the application-oriented compound update operation being defined, and $<op_{ij}>$, is another application-oriented compound update operation or one of the compound update operations, 'insert', 'delete', or 'modify', defined above.

An application oriented compound update operation has one of the following forms:

- <operation_name>(<relation_name>(<tuple spec>))
- <operation_name>(<parameter>)

where <operation_name> is the name of the application-oriented compound update operation, and <parameter> is a tuple of <domain variable>s.

The semantics is as defined for the compound update operations.

It is a matter of taste whether these application-oriented compound update operations should be defined only in terms of the compound 'insert', 'delete', and 'modify' operations above, or whether primitive update operations can be used too. In the first case, the user can be allowed to define the application-oriented operations, whereas, in the latter case, only the database designer should be allowed to define them. In our definition above we have chosen the first alternative, and have thereby introduced a strict hierarchy.

$$\begin{tikzcd}
\text{application-oriented compound update operations} \\
\text{compound update operations} \arrow[u] \\
\text{primitive update operations} \arrow[u]
\end{tikzcd}$$

We envision the compound update operations to be specified in the conceptual schema and the application-oriented compound update operations to be specified in external schemata of a database.

The application-oriented compound update operations support two alternative kinds of external views on a database. Corresponding to the two syntactical forms above; they are:

- **a relation independent operation-oriented view, and**
- a relation dependent view.

Only the relation dependent update operations will be used in the rest of this paper.

Because the application-oriented compound update operations are defined in terms of the compound update operations on conceptual schema relations, view updates are fully supported. The update dependencies will prompt the user for data wherever needed to resolve ambiguities.

The formalism for update dependencies is very flexible and can distinguish several cases. As any other formalism it takes some practice getting used to, and we therefore give a few guidelines below.

An update dependency of the form

```
<op> → <cond>.
```

is often used as part of an update operation specification to test if the database is already in the state which the user intends to bring it in by invoking <op>. In that case <cond> evaluates to true and <op> succeeds without changing the database state. As an example consider

```
delete(r(T))
→    ¬(r(T)).
→    r(T),
     remove(r(T)).
```

Several alternative implications of an update operation, any of which may be chosen under a certain condition, are specified as follows:

```
<op_1>
→    <cond_1>,
     <op_2>.
→    <cond_1>,
     <op_3>.
```

If the choice of alternative depends on the condition, we specify them as follows:

```
<op_1>
→    <cond_2>,
     <op_2>.
→    <cond_3>,
     <op_3>.
```

where <cond_2> and <cond_3> depend on the parameter for <op_1> and on the database state.

A sequence of implications of an update operation is specified as follows:

```
<op_1>
→   <cond>,
    <op_2>,
    <op_3>,
    ..,
    <op_n>.
```

Side-effects on the evaluation of conditions, resulting from the execution of operations, are controlled by sequences of update dependencies. In

```
<op_1>
→   <cond_1> /\ <cond_2>,
    <op_2>,
    <op_3>.
```

the evaluation of <cond_1> and <cond_2> is not affected by <op_2> and <op_3>. In

```
<op_1>
→   <cond_1>,
    <op_2>,
    <op_4>.
<op_4>
→   <cond_2>,
    <op_3>.
```

the evaluation of <cond_2> is affected by <op_2>.

In general, we shall allow the user to call any of the update operations with an actual tuple-parameter which is only partially specified. To control this, a couple of alternatives in an update dependency for a compound update operation will be responsible for producing or requesting from the user any additional values needed for the update operation to be well-defined. As an example consider

```
insert(person(Name,Soc_sec_no))
→   var(Name),
    write('person name?'),
    break,
          read(Name),
          insert(person(Name,Soc_sec_no)).
     →    .
          .
          .
     →    nonvar(Name) /\ nonvar( Soc_sec_no) /\
          ¬(person(_,Soc_sec_no)),
          add(person(Name,Soc_sec_no)).
```

where var(Name) is true if 'Name' is uninstantiated in a call, like insert(person(_,111-22-3333)).

We shall make extensive use of recursive calls with the following two benefits: First, each alternative in a update dependency can concentrate on one aspect of the complete update operation, as in the example above. Second, the recursive calls make the specification of multi-tuple update operations particularly easy, as shown in the example below.

```
     delete(person(Name,Soc_sec_no))
     →    nonvar(Name) /\ var(Soc_sec_no) /\
          person(Name,S),
          remove(person(Name,S)),
          delete(person(Name,Soc_sec_no)).
     →    nonvar(Name) /\ ¬(person(Name,_)).
```

This part of the delete operation on person will delete all persons with a given name provided the call is delete(person(n,_)).

Special attention must be paid to ensure that recursion stops. In the example above we use the classic scheme. In the first update dependency we decrease the number of tuples in relation 'person' by one each time. And, in the second update dependency, we specify the stop-condition.

Special attention must be paid to conditions. In many cases all conditions or two sets of conditions must be mutually exclusive. See the example above.

update dependencies that always lead to failure need not be specified.

Primitive operation calls in the specification of a compound update operation on relation 'r' should only be given on relation 'r'. If operations on relation 's' need to be called from the specification of operations on 'r', it should be done through the compound update operations defined for 's'. Anything else is considered *bad manners*.

All operations which can be invoked by a user and which insert or modify tuples in the database should type-check all parameters. This should be done as part of the conditions in the update dependencies. Operations communicating with the user to obtain data needed by the update dependencies, like operation 'read' above, should likewise type-check the actual parameter values supplied by the user.

Precise messages to the user will in some situations require alternatives in an update dependency to be split into more alternatives. Also explicit representation of update dependencies leading to failure, including an error message just before the fail-operation, would be helpful to the user.

3. Areas of Application

Update dependencies can be used to specify aspects of databases ranging over integrity constraints, transactions, normalization, view maintenance and update, metadata management. We shall discuss each of these issues in the following subsections.

3.1 Traditional Constraints

Although intended for more advanced uses, the update dependency formalism can be used as a traditional tool for constraint specification and enforcement. A simple example of this is allowing an insertion of tuple 'T' in relation 'r' only if the condition 'cond' is true:

insert(r(T)) $\rightarrow$ cond, add(r(T)).

Corrective actions implied by a constraint violation can also be specified. A simple example of this is the following:

delete(s(T)) $\rightarrow$ c, remove(s(T)).

delete(s(T)) $\rightarrow$ $\neg$(c), remove(s(T)), delete(r(T')).

where 'c' is false if the remove operation on 's' violates an integrity rule between 's' and 'r'; e.g., if there is a referential integrity rule:

"r(T') => s(T)", then the condition 'c' would be "$\neg$(r(T'))".

It seems to be fairly easy to automatically generate update dependencies as an implication of a schema definition (we assume a very powerful schema definition language with several constraint types). In the following we give some examples of automatically generated update dependencies for some standard types of constraints.

Example 1

$R(A_1:D_1, \dots ,A_n:D_n)$

insert $R(A_1, \dots ,A_n)$
$\rightarrow$ $\neg(R(A_1, \dots ,A_n))$,
add$(R(A_1, \dots ,A_n))$.
$\rightarrow$ $R(A_1, \dots ,A_n)$.

delete $R(A_1, \dots ,A_n)$
$\rightarrow$ $\neg(R(A_1, \dots ,A_n))$.
$\rightarrow$ $R(A_1, \dots ,A_n)$

remove($R(A_1, \dots ,A_n)$).

In this example we simply check that a relation is a set. If the tuple is not present in the relation then we add it, if it is already present then we do nothing. For deletion, we check if the tuple is in the database or not. If so, then we remove it. We assume that all variables are instantiated in the call.

□

Example 2

```
R(A_1:D_1, ... ,A_n:D_n)
Key: A_1, ... , A_i.

insert R(A_1, ... ,A_n)
   →   ¬(R(A_1, ... , A_i,_, ... ,_)),
       add(R(A_1, ... ,A_n)).
   →   R(A_1, ... , A_i,_, ... ,_).

delete R(A_1, ... ,A_n)
   →   ¬(R(A_1, ... , A_i,_, ... ,_)).
   →   R(A_1, ... , A_i,_, ... ,_),
       remove(R(A_1, ... , A_i,_, ... ,_)).
```

For insertion we check that the key constraint is not about to be violated and proceed by adding the tuple. If the tuple is already then we do nothing. For deletion we check that the tuple is present in the database before we proceed with the removeion. In the insert operation we again assume that all the variables are instantiated in the call. In the delete operation we only assume that the variables of the key are instantiated.

□

Example 3

In the following schema definition B_1 is a foreign key because it is defined over the primary domain D_1.

```
R(A_1:D_1, ... ,A_n:D_n)
Primary key: A_1;
S(B_1:D_1, ... );
```

In this example the insert operation for R is the same as above because we already assume that the variables are instantiated. Or, maybe we want to distinguish between variables not being instantiated and variables having some null value. In that case we want to produce an insert operation for R which checks that the primary key value of R tuples is not null.

Anyway, the important thing to check here is the referential integrity constraint from B_1 to A_1.

```
insert S(B_1, ... ,B_m)
  →  ¬(S(B_1, ... ,B_m)) /\ R(B_1,_, ... ,_),
     add(S(B_1, ... ,B_m)).
  →  ¬(S(B_1, ... ,B_m)) /\ ¬(R(B_1,_, ... ,_)),
     insert(R(B_1,_, ... ,_)),
     add(S(B_1, ... ,B_m)).
  →  S(B_1, ... ,B_m).

delete R(A_1, ... ,A_n)
  →  R(A_1,_, ... ,_) /\ ¬(S(A_1,_, ... ,_)),
     remove(R(A_1,_, ... ,_)).
  →  R(A_1,_, ... ,_) /\ S(A_1,_, ... ,_),
     delete(S(A_1,_, ... ,_)),
     remove(R(A_1,_, ... ,_)).
  →  ¬(R(A_1,_, ... ,_)).
```

The delete operation on relation S is trivial.

□

Example 4

In this example we merely check a post-condition constraint on the database after an operation. If the constraint is violated we undo the operation.

```
R(A_1:D_1, ... ,A_n:D_n)
post_condition: C(R);

insert R(A_1, ... ,A_n)
  →  add(R(A_1, ... ,A_n)),
     check(C(R)).

check(C(R))
  →  C(R).
```

Checking the post_condition on R can of course be done directly if we allow the implications to be a sequence of conditions and implied operations. I sort of like the more restrictive approach with a condition followed by a sequence of implied operations.

The delete operation would be similar to the insert operation.

Checking a pre_condition is trivial.

□

3.2 Transactions

An update dependency is a control abstraction which defines an update operation by grouping together a set of other update operations. It is analogous to the concept of an abstract data type and it is a generalization of the concept of a database transaction. A collection of update dependencies constitutes a production system which helps and guides the user through a database transaction once he has taken the first step.

The advantage of this approach is that it allows the database to encode the knowledge required to guide the user in carrying out his original update request. We have included into the update dependencies variables which get their bindings via a user interaction. The user may be the update originator but it could also be some other user having the ability to provide the necessary values.

This cooperative approach is based on knowledge specific to the application and it is more useful to the user than that of providing a message about a possible set of constraint violations for which he is unaware. The only drawback to this interactive approach is the delays it introduces waiting for a user response (it is so much faster to reject!). For this reason update dependencies must be carefully constructed, thought through and tested out before they are introduced to the database. In many situations the update dependencies need no user interaction at all, and in the situations where user interaction is needed, there are ways of postponing this interaction; e.g. by inserting null-values for missing data values. Having the update dependencies in a declarative form that can be evaluated and tested gives a chance to the database designer and maintainer to avoid costly errors caused by unrealized update dependencies.

Example 5

In this example, we provide a set of update dependencies for operations on the relations in a database. The similarity between the update dependencies and traditional transactions is self evident.

sold

p#	buyer	qty

ordered

p#	supplier	qty

part

p#	pname	qty

```
insert sold$(P,B,Q_1)$
→   part$(P,\_,\_)$,
    delete part$(P,\_,Q_1)$,
    add sold$(P,B,Q_1)$.
→   $\neg$part$(P,\_,\_)$,
    write("part unknown").

insert ordered$(P,S,Q_1)$
→   ordered$(P,S,Q_2)$,
    modify ordered$(P,S,Q_2+Q_1)$.
→   $\neg$ordered$(P,S,\_)$,
    add ordered$(P,S,Q_1)$.

delete part$(P,N,Q_1)$
→   part$(P,N,Q_2)$ $\wedge$ $Q_1<Q_2$,
    modify part$(P,N,Q_2-Q_1)$,
→   part$(P,N,Q_2)$ $\wedge$ $Q_1\geq Q_2$,
```

modify part(P,N,Q_2-Q_1),
insert ordered(P,_,Q_1*3).

3.3 Normalization

Update dependencies can be used to control some of all the well-known problems associated with normalization and lack of normalization.

Example 6

Given the relation R and a **functional dependency**:

$R(X_1, \dots, X_n)$,
FD: $X_i \rightarrow X_j$

We can automatically produce the following update dependency to enforce the functional dependency in the relation.

```
insert $R(X_1,\dots,X_i,\dots,X_j,\dots,X_n)$
  →  $\neg R(\dots,X_i,\dots)$,
     add($R(X_1,\dots,X_i,\dots,X_j,\dots,X_n)$).
  →  $R(\dots,X_i,\dots,X_j,\dots)$,
     add($R(X_1,\dots,X_i,\dots,X_j,\dots,X_n)$).
  →  $R(\dots,X_i,\dots,X,\dots) \wedge \neg(X=X_j)$,
     add($R(X_1,\dots,X_i,\dots,X,\dots,X_n)$),
     write("currently",$X_i$,"has the associated value",X,"not",$X_j$),
     write("the tuple",$X_1,\dots,X_i,\dots,X,\dots,X_n$,"has been inserted").
```

□

Example 7

The **multivalued dependency** $A \twoheadrightarrow B$ (and $A \twoheadrightarrow C$) in the relation R(A,B,C) can be enforced in the following way:

```
insert R(A,B,C)
      (*new A*)
  →   ¬R(A,_,_),
      add(R(A,B,C)).

      (*old A, new B, old C*)
  →   R(A,_,_) /\ ¬R(A,B,_) /\ R(A,_,C) /\ R(A,_,$C_1$) /\ ¬(C=$C_1$),
      add(R(A,B,C)),
      (*insert other old Cs for the new B*)
      insert(R(A,B,$C_1$)).
  →   R(A,_,_) /\ ¬R(A,B,_) /\ R(A,_,C) /\ ¬(R(A,_,$C_1$) /\ ¬(C=$C_1$)),
      add(R(A,B,C)).
      (*no other old Cs to insert for the new B*)

      (*old A, new B, new C*)
```

```
→   R(A,_,_) /\ ¬R(A,B,_) /\ ¬R(A,_,C) /\ R(A,$B_1$,$C_1$) /\ ¬($B_1$=B) /\ R(A,$B_2$,$C_2$) /\ ¬($C_2$=C),
    add(R(A,B,C)),
    (*insert all old Cs for the new B*)
    insert(R(A,B,$C_2$)),
    (*insert the new C for all the old Bs*)
    insert(R(A,$B_1$,C)).

    (*old A, old B, new C*)
→   ¬R(A,B,C) /\ R(A,_,_) /\ R(A,B,_) /\ R(A,$B_1$,_) /\ ¬($B_1$=B) /\ ¬R(A,$B_1$,C),
    add(R(A,B,C)),
    (*insert the new C for all the old Bs*)
    insert(R(A,$B_1$,C)).
→   ¬R(A,B,C) /\ R(A,_,_) /\ R(A,B,_) /\ ¬(R(A,$B_1$,_) /\ ¬($B_1$=B) /\ ¬R(A,$B_1$,C)),
    add(R(A,B,C)).
    (*no more old Bs to insert the new C for*)
```

□

Example 8

One of the problems in normalization theory is the conflict between decomposing a relation into BCNF components and decomposing it into independent components. The problem is described in [Date 86, p381].

Given the relation SJT(S,J,T) with the functional dependencies (S,J) → T and T → J, and the candidate keys (S,J) and (S,T). The relation is in 3. NF, but not in BCNF. The relation suffers from the well-known update anomalies cause by the fact that the determinant T is not a candidate key.

Decomposing the relation into the two BCNF projections ST(S,T) and TJ(T,J) removes the update anomalies, but unfortunately the two relations cannot be independently updated, because the FD (S,J) → T cannot be deduced from the the only FD represented in the projections, namely T → J.

We can automatically generate update dependencies that control the **inter-relational constraint introduced by the decomposition.** We have to guarantee, that for each pair (s,j) of (S,J) there exist a single t∈T in both ST and TJ.

```
insert ST(S,T)
  →   ¬(ST(S,$T_1$) /\ TJ($T_1$,$J_1$) /\ TJ(T,$J_1$) /\ ¬($T_1$=T)),
      add(ST(S,T)).
  →   ST(S,$T_1$) /\ TJ($T_1$,$J_1$) /\ TJ(T,$J_1$) /\ ¬($T_1$=T).
```

Similarly, we have to create an insert operation on TJ enforcing the inter-relational constraint. And, let us not forget that deletions from ST or TJ may violate the inter-relational constraint, so delete operations must be defined too.

We could in fact generalize this to cover any situation where we replace a relation by two of its projections. Given the relation R(A_1, A_2, ... ,A_n) with the functional dependencies

$FD=\{fd_1, fd_2, ...\}$, and given the decomposition of R into $R_1(...)$ and $R_2(...)$ with the functional dependencies $FD_1=\{ ... \}$ and $FD_2=\{ ... \}$, respectively. For every functional dependency $fd \in FD-(FD_1 \cup FD_2)$, we have to create update dependencies that enforce the interrelational functional dependency.

□

Example 9

Another problem in normalization theory is that only some decompositions have the **lossless join property**.

Assume that we want to replace the relation R(A,B,C) with functional dependencies $A \rightarrow B$ and $C \rightarrow B$ by the two projections $R_1(A,B)$ and $R_2(B,C)$. This decomposition is not "good" because B is not the key in any of R_1 and R_2 and we cannot recreate R by joining them - we fall into the **connection trap**. We can however define update dependencies so that we avoid the connection trap.

At the time of decomposition we compute the complement of R wrt. $R_1{*}R_2$:

$$C_R = R_1{*}R_2 \setminus R$$

Now the idea is that we can insert anything we want in R_1 as long as the B-value does not appear in a tuple in R_2. However, if the B-value does occur in R_2, then we have to add to the complement all the extra tuples that would result in the join of R_1 and R_2 (and vice versa for R_2).

```
insert $R_1(A,B)$
  →  $R_1(A,B)$.
  →  $\neg R_2(B,\_) \wedge \neg R_1(A,B)$,
     add($R_1(A,B)$).
  →  $R_2(B,\_) \wedge \neg R_1(A,B)$,
     add($R_1(A,B)$),
     insert($C_R(A,B,\_)$).

insert $C_R(X,Y,Z)$
  →  var(Z) $\wedge \neg(R_2(Y,V) \wedge \neg C_R(X,Y,V))$,
  →  var(Z) $\wedge R_2(Y,V) \wedge \neg C_R(X,Y,V)$,
     add($C_R(X,Y,V)$),
     insert($C_R(X,Y,\_)$).
  →  var(X) $\wedge$ ...
```

It should be noted that these update dependencies guarantee that whenever we join R_1 and R_2 and subtract C_R we get the value of R at the time the decomposition was made no matter which changes have been made to R_1 and R_2 in the meantime. I might seem more appropriate to somehow let the value of R keep up with the changes made to R_1 and R_2, but there is no way we can guess what the value of R would have been had it not been

replaced by its projections.

□

Example 10

In spite of the adverse effects, a user may have good reasons for replacing two 3rd NF relations by their join.

Let $R_1(A_1,A_2)$ and $R_2(A_2,A_3,A_4)$ be two 3rd NF relations with keys $K_1=\{A_2\}$ and $K_2=\{A_2,A_3\}$. (The A_is may be compound attributes.)

Let R be the join of R_1 and R_2 with key $K=K_1 \cup K_2$.

We define the following update dependency to maintain two relations $C_{R|R1}$ and $C_{R|R2}$ such that $R_1=R[A_1,A_2] \cup C_{R|R1}$ and $R_2=R[A_2,A_3,A_4] \cup C_{R|R2}$, and we use these formulas to compute R_1 and R_2 whenever they are referenced.

```
insert $R(A_1,A_2,A_3,A_4)$
  → $\neg(A_1=null) \wedge \neg(A_2=null) \wedge A_3=null \wedge A_4=null \wedge$
    $\neg R(\_,A_2,\_,\_)$,
    insert($C_{R|R1}(A_1,A_2)$.
  → $\neg(A_1=null) \wedge \neg(A_2=null) \wedge A_3=null \wedge A_4=null \wedge$
    $R(\_,A_2,\_,\_)$.
  → $A_1=null \wedge \neg(A_2=null) \wedge \neg(A_3)=null) \wedge \neg(A_4=null) \wedge$
    $\neg R(\_,A_2,A_3,\_)$,
    insert($C_{R|R2}(A_2,A_3,A_4)$.
  → $A_1=null \wedge \neg(A_2=null) \wedge \neg(A_3)=null) \wedge \neg(A_4=null) \wedge$
    $R(\_,A_2,A_3,\_)$.
  → $\neg(A_1=null)\ \neg(A_2=null) \wedge \neg(A_3)=null) \wedge \neg(A_4=null) \wedge$
    $\neg R(\_,A_2,A_3,\_)$,
    add($R(A_1,A_2,A_3,A_4)$).
  → $\neg(A_1=null)\ \neg(A_2=null) \wedge \neg(A_3)=null) \wedge \neg(A_4=null) \wedge$
    $R(\_,A_2,A_3,\_)$.
```

This update dependency avoids the update problems associated with **nonfull functional dependencies created by joins.**

Similarly, we could specify update dependencies avoiding the problems associated with **transitive dependencies created by joins.**

3.4 Views

It is well-known that views can often not be updated because the view definition does not provide enough information for the system to decide what the corresponding update(s) on the base relations should be.

An excellent way of providing more information at view definition time is suggested in [Keller 86]. The idea is to provide the user with a choice of one among a set of alternative

view update policies provided by the system. The resulting update policy could easily be specified in terms of update dependencies.

Using update dependencies at view definition time to describe the implications of a view update on the corresponding base relations is a general solution to the view update problem.

This approach however highlights another problem, namely that we often do not know what we want the corresponding updates on the base relations to be. The update dependencies allow us to specify anything we want, but they cannot help us if we don't know what we want! The situations where we do know what we want are often those where certain policies from real life are enforced.

Example 11

Assume that we have the following base relations:

```
EMPLOYEE(SSN, NAME, DEPARTMENT, DATE_HIRED)
WAITING_LIST(SSN, NAME, DATE_ENTERED)
```

and the view:

```
#EMPLOYEE_DEPARTMENT(DEPARTMENT, #EMPL)=
select DEPARTMENT, count(SSN)
from EMPLOYEE
group by DEPARTMENT
```

Now, assume that we have the following policies:

New employees are hired by the company in the order they appear on the waiting list: first entered, first hired.

Employees are fired from departments following the principle: last hired, first fired.

modify #EMPLOYEE_DEPARTMENT(D,N)
    (**computes the view to see if department D already has the desired size N**)
→ #EMPLOYEE_DEPARTMENT(D,N).

    (**bring department D down to N employees**)
→ #EMPLOYEE_DEPARTMENT(D,N_1) /\ N_1>N / EMPLOYEE(SSN_1, $NAME_1$, D, $DATE_1$)
    /\ ¬(EMPLOYEE(SSN_2, $NAME_2$, D, $DATE_2$) /\ $DATE_2$>$DATE_1$),
    delete(EMPLOYEE(SSN_1, $NAME_1$, D, $DATE_1$)),
    modify(#EMPLOYEE_DEPARTMENT(D,N)).

    (**bring department D up to N employees**)
→ #EMPLOYEE_DEPARTMENT(D,N_1) /\ N_1<N /\
    WAITING_LIST(SSN_1, $NAME_1$, $DATE_1$) /\ ¬(WAITING_LIST(SSN_2, $NAME_2$, $DATE_2$)
    /\ $DATE_2$<$DATE_1$),
    delete(WAITING_LIST(SSN_1, $NAME_1$, $DATE_1$)),
    insert(EMPLOYEE(SSN_1, $NAME_1$, D, current_date)),

modify(#EMPLOYEE_DEPARTMENT(D,N)).

□

A view definition consists of two parts:

- an intension definition, and
- an extension definition.

The intension definition consists of two parts:

- a relation name and a set of attribute names that together carry the meaning of the relation, and
- a set of compound update operations that define and control all operations on the extension of the view.

The extension definition defines the extension of the view in terms of the extensions of base relations and other views.

The compound update operations on views are defined in terms of implied non-primitive compound update operations on base relations and other views; and an actual view update is executed through the primitive operations on the base relations implied by the compound update operations on the base relations.

The compound update operations on views define a mapping from the views to the base relations. This mapping is **missing** in the relational model where views are only extensionally defined by a mapping from the base relations to the views. This means that our approach supports **updating through views**. To resolve ambiguity of a view update, the compound update operations on the view may have to request additional data from a user. If the additional data is outside of the view of the user who initiated the update, then another user may have to supply it. Additional data will only be needed in situations where traditional approaches to view update would fail.

A **view-update-check** is an important part of a compound update operation on a view. It guarantees, that if a view update succeeds, then the extension of the view will be as intended by the user who initiated the view update. We regard the incorporation of a view-update-check in a compound update operation on a view as a programming discipline put on the database designer. In some situations we actually have to accept view updates that do not obey the view-update-check.

We can express the extensional part of a view definition by update dependencies as follows:

v(T) --> C.

where 'v' is the name of the view, 'T' is a tuple variable for 'v', and 'C' is a condition.

We illustrate this technique for the operators of the relational algebra below.

Union

v(T) --> ¬(¬(s(T) /\ ¬(r(T))).

Intersection

v(T) --> s(T) /\ r(T).

Minus

v(T) --> s(T) /\ ¬(r(T)).

Times

$v(A_1, \ldots ,A_{m+n})$ -->
$s(A_1, \ldots ,A_m)$ /\ $r(A_{m+1}, \ldots ,A_{m+n})$.

Selection

v(T) --> s(T) /\ condition.

Projection

$v(A_1, \ldots ,A_m)$ --> $s(A_1, \ldots ,A_m,\_, \ldots ,\_)$.

Join

$v(A_1, \ldots ,A_k,B_1, \ldots ,B_m,C_1, \ldots ,C_n)$ -->
$s(A_1, \ldots ,A_k,B_1, \ldots ,B_m)$ /\ $r(B_1, \ldots ,B_m,C_1, \ldots ,C_n)$.

Divide

Exercise!

Example 12
We shall now give an example of a view definition following our approach.

Given:
base relation 'people' and two compound update operations 'insert' and 'delete'.

people

name	addr	age

```
insert(people(N,A,Y))
-->   nonvar(N) /\ nonvar(A) /\ var(Y),
      write('age?'),
      break,
      read(Y),
      add(people(N,A,Y)).
```

```
delete(people(N,A,Y))
-->   .
      .
      remove(people(N,A,Y)).
```

The definition of a view 'teenagers' consists of:

Intension:

teenagers

name	addr

```
insert(teenagers(N,A))
-->   teenagers(N,A).
-->   ¬(teenagers(N,A)) /\ ¬(people(N,A,_)),
      insert(people(N,A,_)),
      insert(teenagers(N,A)).
```

```
delete(teenagers(N,A))
-->   ¬(teenagers(N,A)).
-->   teenagers(N,A),
      delete(people(N,A,_)),
      delete(teenagers(N,A)).
```

Extension:

```
teenagers(N,A)
-->   people(N,A,Y) /\ (12<Y<20).
```

The example is very simplified. For the base relation 'people' we indicate two compound update operations 'insert' and 'delete'. These are defined in turn from primitive update operations 'add' and 'remove'. For the 'insert' operation we have indicated the situation where 'age' is not instantiated and therefore requested from the user.

The intensional definition of the view relation 'teenagers' consists of the definition of the names for the relation and its attributes. And, it includes the definition of the two compound view update operations.

Both the intensional and the extensional part of the view definition must be given before any update or retrieval to the view takes place. Why this is the case, will be explained later.

The compound view insert operation succeeds if the tuple is already present in the view. To test this, we need the extension definition as claimed above. If the tuple is not present, then we call insert on 'people' with the 'age' uninstantiated, and the system will try to make this operation succeed, and it will request the missing age from the user in the process of doing so. If the insertion in people succeeds, then we are back in the insertion in 'teenagers', where we make a recursive call. The purpose of this recursive call is to check if the inserted tuple is now visible through the view 'teenagers'. This recursive call plus the condition is what wer have termed a view-update-check. If this check fails after the insertion in 'people' succeeded,

then it must be because the user gave an age which does not qualify the person as a teenager. The whole operation must therefore fail and the insertion in people will be undone. If the check succeeds, that is, the user can through the view see the intended result of his insertion, then the whole operation succeeds.

Without the view-update-check, not only would it be strange that the user can not see the result of what he is doing, but he would also be able to insert something which is not part of his view, namely people who are not teenagers.

In the delete operation on 'teenagers' we use a view-update-check too; the user can only delete things that are visible through his view.

Please note the strict hierarchy between compound update operations on views, compound update operations on base relations, and primitive update operations on base relations.

□

Updates to base relations will always be reflected in a view defined on the base relations if we recompute the view. However, recomputing a view every time one of its base relations is updated may be a very high price to pay. In [Roussopoulos] techniques are presented for maintaining a materialized view during updates to base relations.

An interesting idea is to include implied operations on materialized views in the specification of operations on base relations. Ideally, these implied operations would be inserted automatically in the specification of the operations on the base relations at view definition time.

Example 13

The update dependencies in this example specify that a materialized views is to be maintained during updates of the base relations it is derived from.

Base relations:

R(A,B) and S(B,C)

View:

T(A,B,C) = R(A,B) join S(B,C)

```
insert R(A,B)
  →  ¬R(A,B),
     add(R(A,B)),
     insert(R(A,B).

  →  R(A,B) /\ S(B,C) /\ ¬T(A,B,C),
     add(T(A,B,C)),
     insert(R(A,B)).

  →  R(A,B) /\ ¬(S(B,C) /\ ¬T(A,B,C)).
```

□

3.5 Metadata Management

In this section we give a comprehensive example of the use update dependencies by defining the compound update operations in a meta-schema for the relational data model.

The meta-schema defines and controls all operations on schemata defined in terms of the relational data model.

The meta-schema is itself defined in terms of the relational data model, and a copy of the meta-schema is explicitly stored as part of its own extension.

The beauty of this "stored self-describing meta-schema approach" is, that not only does it break the infinite chain of meta-levels, it also allows us to use existing DML-processors to retrieve and change schema definitions. This is illustrated in Figure 1.

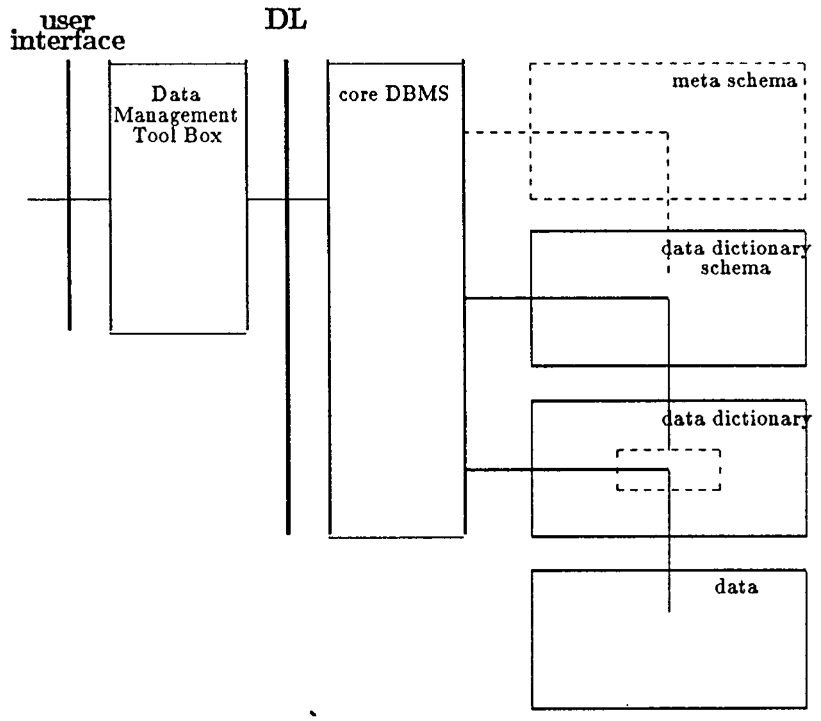


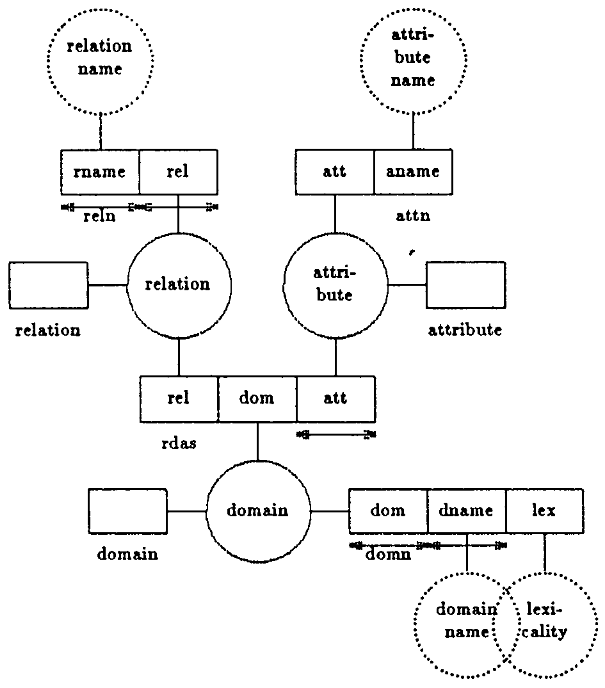
Figure 1.

For more information on self-describing meta-schema and databases we refer to [3,7].

3.5.1 The Meta-Schema

The part of the meta-schema we shall concentrate on in this example is defined in Figure 2.

Figure 2. Core 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. There are several constraints not represented above: reln [relation] = relation; [relation] $\supset$ rdas [relation]; domn [domain] = domain; [domain] $\supset$ rdas [domain]; rdas [attribute] = attribute, [attribute] $\supset$ attn [attribute]; also atttribute names must be unique within relations.

All these constraints and more will be specified in the update dependencies below.

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 3. (We have omitted the unary relations: 'relation', 'domain', and 'attribute'.)

reln

rname: relation name	rel: relation
reln	r_1
domn	r_2
attn	r_3
rdas	r_4

domn

dom: domain name	dname: domain	Lex: lexicality
d_1	relation	non-lexical
d_2	attribute	non-lexical
d_3	domain	non-lexical
d_4	relation name	lexical
d_5	attribute name	lexical
d_6	domain name	lexical
d_7	lexicality	lexical

rdas

reln: relation	dom: domain	att: attribute
r_1	d_4	a_1
r_1	d_1	a_2
r_2	d_3	a_8
r_2	d_6	a_9
r_2	d_7	a_{10}
r_3	d_2	a_3
r_3	d_5	a_4
r_4	d_1	a_5
r_4	d_3	a_6
r_4	d_2	a_7

attn

att: attribute	aname: attribute name
a_1	rname
a_2	rel
a_3	att
a_4	aname
a_5	rel
a_6	dom
a_7	att
a_8	dom
a_9	dname
a_{10}	lex

Figure 3.

3.5.2 The update dependencies

We shall now use update dependencies to specify the following compound update operations in the meta-schema, leaving out the modify operations.

Please note, that we do not in this example consider the important problem of propagating schema changes to data.

	insert	delete
domain		
domn		
relation		
reln		
attribute		
attn		
rdas		

Figure 4.

Insert Operations

```
insert(domn(D,N,L))
→   var(D),
    new(domain(D)),
    insert(domn(D,N,L)).
→   var(N),
    get('domain name',N),
    insert(domn(D,N,L)).
→   var(L),
    get('lexicality',L),
    insert(domn(D,N,L)).
→   nonvar(D) /\ nonvar(N) /\ nonvar(L) /\
    ¬(domn(D,-,-))/\ ¬(domn(-,N,-)),
    add(domn(D,N,L)),
    insert(domain(D)).
→   nonvar(D) /\ nonvar(N) /\ nonvar(L) /\ domn(N,D,L).
```

The first three update dependencies in this compound update operation specification simply serve to fully determine the tuple to be inserted in relation 'domn'. If the domain surrogate is not specified in the call the system creates one using the primitive operation 'new'. If domain name or lexicality are not specified the user is prompted for the values.

If or when all parameters are determined there are three possibilities. If the tuple is already present in relation 'domn' the operation succeeds with the database state unchanged. If neither the domain surrogate nor the name is used in relation 'domn' we add the tuple and propagate the update by a call of an insertion in relation 'domain' of the new domain surrogate. If the domain surrogate and/or name are present in relation 'domn', but not in the same tuple the operation fails. The last of these three possibilities is taken care of by not specifying an update dependency for it; see the definition of update dependencies. We could have specified an update dependency with a condition evaluating to true in exactly this situation followed by a call of the primitive operation 'fail', which doesn't change the database state and always fails.

```
insert(domain(D))
→   var(D),
    new(domain(D))
    insert(domain(D)).
→   nonvar(D) /\ domain(D).
→   nonvar(D) /\ ¬(domain(D)) /\ ¬(domn(D,_,_,)),
    add(domain(D)),
    insert(domn(D,_,_)).
→   nonvar(D) /\ ¬(domain(D)) /\ domn(D,_,_),
    add(domain(D)).
```

If the domain surrogate to be inserted is uninstantiated in the call the system creates a new one and proceeds with the insertion. This is not in conflict with insertions in 'domn' because a call of insertion in 'domain' issued from 'domn' will always instantiate the domain surrogate. It will therefore be the same domain surrogate which is inserted both places.

If the domain surrogate value is known in the call and is already present in relation 'domain' the operation succeeds with the database state unchanged. If the domain surrogate value is known and not present in relation 'domain' it is added; and if the domain is not recorded to have a name we call an insertion on relation 'domn' to take care of this.

```
insert(relation(R))
→   var(R),
    new(relation(R)),
    insert(relation(R)).
→   nonvar(R) /\ relation(R).
→   nonvar(R) /\ ¬(relation(R)),
    add(relation(R)),
    insert(reln(-,R)).
```

If the variable R is uninstantiated when the insertion into relation 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 a name we propagate by triggering an insertion into the relation 'reln' of the tuple (-,R) indicating that relation_name is at this point unknown.

```
insert(reln(N,R))
```

```
→   var(R),
    new(relation(R)),
    insert(reln(N,R)).
→   var(N),
    get('relation name', N),
    insert(reln(N,R)).
→   nonvar(N) /\ nonvar(R) /\ reln(N,R).
→   nonvar(N) /\ nonvar(R) /\ ¬(reln(N,-)) /\ ¬(reln(-,R)),
    add(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 add the tuple and propagate by inserting R in relation 'relation'. Note that 0-ary relations are allowed, that is relations without attributes and extension are allowed. Any other choice could just as easily have been specified.

```
insert(attribute(A))
→   var(A)),
    new(attribute(A)),
    insert(attribute(A)).
→   nonvar(A) /\ attribute(A).
→   nonvar(A) /\ ¬(attribute(A)),
    add(attribute(A)),
    insert(attn(A,_)),
    insert(rdas(_,_,A).
```

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

```
insert(attn(A,N))
→   var(A),
    new(attribute(A)),
    insert(attn(A,N)).
→   var(N) /\ nonvar(A) /\ ¬(attn(A,_)),
    get('attribute name', N),
    insert(attn(A,N)).
→   var(N) /\ nonvar(A) /\ attn(A,_).
→   nonvar(A) /\ nonvar(N) /\ attn(A,N).
→   nonvar(A) /\ nonvar(N) /\ ¬(attn(A,N)) /\ ¬(attn(B,N) /\
        rdas(R,_,B) /\ rdas(R,_,A) /\ (B≠A)),
        add(attn(A,N)),
        insert(attribute(A)),
        insert(rdas(_,_,A)).
```

The first update dependency produces an attribute surrogate if needed. If the attribute already has a name a new one will not be requested if missing in the call. Finally, if at this point the uniqueness of attribute names in relations is not about to be violated the name is added in attn, and propagation calls to 'attribute' and 'rdas' are made.

```
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)),
    get('relation surrogate', R),
    insert(rdas(R,D,A)).
→   var(D) /\ ¬(nonvar(A) /\ rdas(_,_,A)),
    get('domain surrogate', 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)),
    add(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 the tuple is added in 'rdas'. This may cause propagation to 'relation' and 'domains' which is taken care of; and, finally, propagation to 'attribute' is called.

Delete Operations

In all the insertion operations above we have been very flexible wrt. the use of uninstantiated variables in operation calls. For the deletion operations below we shall limit this practice somewhat, not because we can't specify the operations that way, but because we don't want to. The reason therefore is that some deletions with uninstantiated variables would have too big an effect, e.g., delete(domain(_)) would delete all domains.

```
delete(domain(D))
```

```
→   var(D),
    get('domain surrogate', D),
    delete(domain(D)).
→   nonvar(D) /\ ¬(domain(D)).
→   nonvar(D) /\ domain(D) /\ ¬(protected_domain(D)),
    remove(domain(D)),
    delete(domn(D,_,_)),
    delete(rdas(_,D,_)).
```

If the variable D is unistantiated in the call the system prompts the user for the value. If the domain 'D' is not present the operation succeeds with the database unchanged. If the domain 'D' is present and not protected from deletion we remove it and propagate to deleteions in 'domn' and 'rdas'. The protected domains are those in the stored meta-schema copy.

```
delete(domn(D,N,L))
→   var(D) /\ var(N),
    get('domain_name', N),
    delete(domn(D,N,L)).
→   nonvar(D) /\ var(N) /\ ¬(domn(D,_,_)).
→   var(D) /\ nonvar(N) /\ ¬(domn(_,N,_)).
→   nonvar(D) /\ nonvar(N) /\ ¬(domn(D,N,_)).
→   nonvar(D) /\ var(N) /\ domn(D,_,_) /\
    ¬(protected_domain(D)),
    remove(domn(D,_,_)),
    delete(domain(D)).
→   var(D) /\ nonvar(N) /\ domn(E,N,_) /\
    ¬(protected_domain(E)),
    remove(domn(E,N,_)),
    delete(domain(E)).
→   nonvar(D) /\ nonvar(N) /\ domn(D,N,_) /\
    ¬(protected_domain(D)),
    remove(domn(D,N,_)),
    delete(domain(D)).
```

To delete from domn we must know either D or N, the domain surrogate or name. If we only know one of the two and no tuple containing it exist in domn we accept with the database state unchanged. If we do know both, but no tuple in domn contain both we also succeed with the database state unchanged. It is a reasonable precaution to require both surrogate and name of the domain to match if the user has supplied both. Using that same precaution the following three rules in the above allow deletion on the surrogate alone, or the name alone, or both together. In all three situations, if the domain is not one of the protected ones, we removethe one tuple from 'domn' and propagate the deletion to relation 'domain'.

```
delete(rdas(R,D,A))
→    ¬(rdas(R,D,A)).
→    var(A) /\ var(D) /\ var(R),
     write('nothing done').
→    nonvar(A) /\ rdas(R,D,A) /\ ¬(protected_attribute(A)),
     remove(rdas(R,D,A)),
     delete(attribute(A)).
→    nonvar(D) /\ var(A) /\ var(R) /\ rdas(R,D,A) /\
     ¬(protected_domain(D)),
     remove(rdas(R,D,A)),
     delete(attribute(A)),
     delete(rdas(_,D,_)).
→    nonvar(R) /\ var(A) /\ var(D) /\ rdas(R,D,A) /\
     ¬(protected_relation(R)),
     remove(rdas(R,D,A)),
     delete(attribute(A)),
     delete(rdas(R,_,_)).
→    nonvar(R) /\ nonvar(D) /\ var(A) /\ rdas(R,D,A) /\
     ¬(protected_relation(R)),
     remove(rdas(R,D,A)),
     delete(attribute(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, and since we want relations to be allowed to exist without any attributes (that is, 0-ary relations) the only propagation from deleting tuples from 'rdas' x is to 'attribute' and 'attn'. The decisions above are reasonable from the point-of-view of the DB-designer, because it allows him to also use the schema DB as a list of what he must remember to complete during the design.

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

Strictly, a proof for the above should be given, but the recursion scheme used is classic: all recursive calls are at the bottom of each alternative of the update dependency; one tuple is removed each time and none are inserted; finally, the first rule is the stop condition. It couldn't be more elegant!

Note, that it is recursion that allows the multiple tuple deletion. Had we tried to enforce multiple deletions through backtracking enforced by repeated failures we would have gotten in conflict with that part of our system, which undoes operations already done during an attempt to succeed. **This is important.**

And, now the explanation of the operation. If no tuples matching the actual parameter is present in 'rdas' the operation succeeds with the DB 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:		r		r	r		r
D:			d	d		d	d
A:	a	a	a	a			

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 only the domain-surrogate is give: rdas(_,d,_), we must delete all uses of that domain in any relation and propagate each deletion to 'attributes'. If only the relation-surrogate is given: rdas(r,_,_), all attributes for that relation is deleted frorm 'rdas' and propagated to deletions in 'attributes'. Finally, if both relation and domain-surrogate are given: rdas(r,d,_), we must delete all uses of the given domain in the given relation, and propagate to 'attribute'. The last, but two, rule obviously would be matched by a previous domain deletion. And, the last, but one by a previous relation deletion.

```
delete(attribute(A))
→   ¬(attribute(A)).
→   nonvar(A) /\ attribute(A) /\ ¬(protected_attribute(A)),
    remove(attribute(A)),
    delete(rdas(_,_,A)),
    delete(attn(A,_)).
→   var(A),
    get('attribute surrogate', A),
    delete(attribute(A)).
```

If the attribute is not present in 'attribute' the operation succeeds with the DB 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' were move 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', 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(attn(A,N))
→   ¬(attn(A,N)).
→   var(A) /\ var(N),
    write('nothing done').
→   nonvar(A) /\ var(N) /\ attn(A,N) /\
    ¬(protected_attribute(A)),
    remove(attn(A))
    delete(attribute(A)),
    delete(attn(A,_)).
→   var(A) /\ nonvar(N) /\ attn(A,N) /\
    ¬(protected_attribute(A)) /\ attn(A,M) /\ ¬(M=N),
    remove(attn(A,N)),
    delete(attn(_,N)).
→   var(A) /\ nonvar(N) /\ attn(A,N) /\
    ¬(protected_attribute(A)) /\ ¬(attn(A,M) /\ ¬(M=N)),
    remove(attn(A,N)),
    delete(attribute(A)),
    delete(attn(_,N)).
→   nonvar(A) /\ nonvar(N) /\ attn(A,N) /\
    ¬(protected_attribute(A)) /\ attn(A,M) /\ ¬(M=N),
    remove(attn(A,N)).
→   nonvar(A) /\ nonvar(N) /\ attn(A,M) /\
    ¬(protected_attribute(A)) /\ ¬(attn(A,M) /\
    ¬(M=N)),
    remove(attn(A,N)),
    delete(attribute(A)).
```

Deletion from 'attn' follow the same pattern: If only the attribute is given, all names are deleted. If only a name is given, all attribute having only this name are deleted, whereas those having other names left remains. (5th and 4th update dependency, respectively.) The last update dependency, but one, if both name and attribute are given, but the given attribute has other names we just remove this one name. In the last update dependency, if the name was the only name, we must, in addition, remove the attribute.

Again, we must verify that delete(attn( )) and delete(attribute( )) work correctly. And, that recursion within delete(attn( )) is as we want it.

The latter first; all update dependencies removeone element from 'attn' or stops. All recursive calls have a matching update dependencies. Recursive calls follow either a certain name or attribute. All recursive calls follow a one tuple removeion.

All calls to delete an attribute, except the first for each attribute, returns because of the first update dependency in delete(attribute( )). The first call for deleting an attribute, actually returns a call to delete(attn( )). Therefore there may at some point be two identical delete(attn( )), but the first and last of these will stop on the first update dependency of delete(attn( )), if the job is already done.

```
delete(relation(R))
→   ¬(relation(R)).
→   var(R),
    get('relation surrogate', R),
    delete(relation(R)).
→   nonvar(R) /\ relation(R) /\ ¬(protected_relation(R)),
    remove(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'. Since deletions form 'rdas' does not backfire to 'relation' we only have to verify calls between 'relation' and 'reln'.

```
delete(reln(N,A))
→   ¬(reln(N,R)).
→   var(N) /\ var(R),
    get('relation name', N),
    delete(reln(N,_)).
→   ¬(var(N) /\ var(R)) /\ reln(N,R) /\
    ¬(protected_relation(R)),
    remove(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 deletion(reln( )) stops on the first rule.

We have now given a complete specification of insertion and deletion operations in the core meta-schema for the relational data model. The two compound update operations defined for each meta-schema relation guarantee that the extension of the meta-schema at all times can be interpreted as a correctly defined relational schema.

4. The Interpreter

update dependencies, once specified, need to be thoroughly tested and validated. The logic programming language Prolog is a convenient tool to implement update dependencies. However, it should be pointed out that despite certain similarities between update dependencies and logical implications, update dependencies are not part of any formal logical system. The update dependency language is a practical tool for specifying database updates.

Prolog's treatment of database updates is based on ad hoc "extra logic" primitives which are inadequate for update dependencies as we shall demonstrate. We propose extensions to Prolog to include programming language primitives for implementing update dependencies.

The main update primitives in Prolog are "assert" and "retract". A main problem with the definition of assert and retract from a database point-of-view is that their effects are not undone during backtracking. For example consider the following program.

```
insert(t(X,Y))
:-   pre_condition,
     assert(t(X,Y)),
     post_condition.
```

This rule specifies a database insertion operation of the tuple t(X,Y) into the database. From a database point-of-view, the operation should work the following way: First, the constraints in the pre_condition are checked. If the pre_condition is true, then the tuple t(X,Y) is asserted and the constraints in the post_condition are checked to determine whether the updated database satisfies them. If not, then the inserted tuple violates the integrity constraints, and the assertion should be undone.

But, this is not the way Prolog works, the bad tuple will remain in the database! Simply undoing assertions and retractions when backtracking is not sufficient either as shown by the following example:

```
change(t(X,Y),t(X,Z))
:-   pre_condition,
     t(X,Y),
     compute(Y,Z),
     retract(t(X,Y)),
     assert(t(X,Z)),
     post_condition.
```

This example illustrates an update of tuple t(X,Y) to tuple t(X,Z). Suppose, after the retraction and assertion, the post_condition fails. If we assume that the retraction and the assertion are undone during backtracking, then tuple t(X,Z) will be retracted and tuple t(X,Y) will be re-asserted back into the database. However, when asserting a tuple, Prolog appends it to the end of the database. Physically the new t(X,Y) may be in a different location from

the previous t(X,Y). In this case, goal t(X,Y) will resatisfy because Prolog only looks for physical tuples rather than logical tuples, and the program will go into an infinite loop. So, care must be taken in undoing retractions and assertions. The undo operation must make sure that the physical location of the tuples will not change during backtracking.

Since assert and retract can be used for other general programming purpose, we propose to define three primitives for the purpose of database updates. They are: add, remove, and modify.

Predicate add(X) has one argument: the tuple to be added into the database. X is restricted to tuples only (unlike assert which allows a general rule to be asserted). Add will add the tuple to the end of the database. Upon backtracking, the added tuple will be removed from the database.

Predicate remove(X) also has one argument: the tuple to be removed from the database. X again is restricted to tuples only. When executed, remove will mark the tuple being removed so that later on it will not be used in the proof. However, the tuple being removed will not physically be removed from the database until the top level goal succeeds. If backtracking occurs, the mark on this tuple will simply be erased and the fact will remain unchanged both physically and logically.

Predicate modify(X,Y) has two arguments: the tuple X to be modified and the tuple Y which is the result of modifying X. Again, X and Y are both restricted to tuples only. Logically, modify(X,Y) is equivalent to remove(X) followed by add(Y), but physically it is different. This is important to Prolog because the order of facts in Prolog is crucial. The physical location of the tuple should not be changed. If we had used remove(X) followed by add(Y), then the physical location would have been changed because add(Y) will always add to the end of the database. We want modify(X,Y) to make an in place modification.

Execution of modify is as follows: first the address of tuple X is saved. Then tuple X is modified to Y inplace. If backtrack occurs then X will simply be copied back to Y.

We have now defined the three database update primitives for Prolog. The main point is that both remove and modify will not only undo the operation logically during backtracking, they will also guarantee that the physical locations of the involved tuples are unchanged as well.

The implementation of the other primitives described earlier in this paper were straightforward. However, we still have to include code that prevents surrogates created by the primitive 'new' from ever being visible to the user.

In general, Prolog fits our update dependency ideas very well with its backtracking, rule matching, variable binding and other mechanisms. Also Prolog has proved to be very flexible to accommodate changes.

5. Research Issues

We are looking for elements of an "algebra" for conditional statements.

Suppose we have two relations R(A1,A2) and S(A2). The first operation below enforces a key constraint on A1 of R, and the second enforces a referential integrity constraint from A2 in R to A2 in S:

```
insert(R(a1,a2))                    insert(R(a1,a2))
  →   R(a1,_).                        →   S(a2),
  →   not(R(a1,_)),                       assert(R(a1,a2)).
      assert(R(a1,a2)).               →   not(S(a2)),
                                          assert(S(a2)),
                                          assert(R(a1,a2)).
```

The "algebra" for update dependencies should make the system capable of combining independently automatically produced update dependencies into one update dependency. The one definition of the semantics of this resulting update dependency that makes sense so far, is that the invariant it obeys should be the conjunction of the invariants obeyed by the component update dependencies. However, this is not an operational specification, as we want.

Another issue for further research is operational abstractions of the update dependency formalism that make the specification language more user-friendly. This issue seems to have fairly nice solutions. Next, we illustrate a couple of (very) informal examples.

```
while E do Stmt;
op_1(t)
  →   E,
      Stmt,
      op_1(t).
  →   ¬E.


repeat Stmt until E;
op_1(t)
  →   Stmt,
      check(E).
check(E)
  →   E.
  →   ¬E,
      op_1(T).


case E of
                e_1:   Stmt_1;
                       ...
                       ...
                e_n:   Stmt_n;
 otherwise: Stmt_o;
end;


op_1(t)
  →   E=e_1,
      Stmt_1.
  →   ...
      ...
  →   E=e_n,
    →   $\neg(E=e_1) \wedge \dots \wedge \neg(E=e_n)$,
        $Stmt_o$.


if E then $Stmt_1$ else $Stmt_2$;

$op_1(t)$
    →   E,
        $Stmt_1$.
    →   $\neg E$,
        $Stmt_2$.
```

References

[1] Breutman, B., Falkenberg, E., Mauer, R., "CSL, A Language for Defining Conceptual Schemas," in G. Gracchi, G.M. Nijssen (ed) "Database Architecture" North Holland Publishing Company, 1979.

[2] Brodie, M.L., "On Modelling Behavioural Semantics of Data", Proc. 7th International Entity-Relationship Conference, Cannes, France, Sept. 1981.

[3] 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, 1984.

[4] Gray, J.N., "Notes on Database Operations", IBM Technical Report RJ2188, San Jose, California, 1978.

[5] Hammer, M., and McLeod, D., "A Framework for Database Semantic Integrity", Proc. 2nd International Conference on Software Engineering, San Francisco, California, Oct. 1976.

[6] Jarke, M., Clifford, J., and Vassiliou, Y., "An Optimizing PROLOG Front-end to a Relational Query Systems", Proc. of ACM SIGMOD Conference on Management of Data, June, 1984.

[ ] Keller, A *** View Update policies ***

[7] 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, April 16-18, 1985.

[8] Mylopoulos, J., Bernstein, P.A., and Wong, H.K.T., "A Language Facility for Designing Interactive Database-Intensive Applications", ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980, pp. 185-207.

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

[10] Roussopoulos, N., and Mark, L., "A Self-Describing Meta-Schema for the RM/T Data Model", IEEE Workshop on Languages for Automation, Chicago, November 1983.

[11] Shepherd, A., and Kerschberg, L.,"PRISM: A Knowledge Based System for Semantic Integrity Specification and Enforcement in Database Systems", Proc. of ACM SIGMOD Conference on Management of Data, June, 1984.

[12] Stonebraker, M., Anderson, E, Hanson, E., "QUEL as a Data Type", Proc. of ACM SIGMOD Conference on Management of Data, June, 1984.

[13] Tsur, K., and Zaniolo, C., "An Implementation of GEM - Supporting a Semantic Data

Model on a Relational Back-end", Proc. of ACM SIGMOD Conference on Management of Data, June, 1984.

[14] Wiederhold, G., and El-Masri, R., "A Structural Model for Database Systems", Computer Science Department, Stanford University, Technical Report STAN-CS-79-722, Stanford, California, 1979.

[15] Yeh, R.T., Roussopoulos, N., and Chu, B., "Reusable Software Management", To Proceedings for the IEEE COMPCON, Arlington, VA, Sept. 1984.

[16] Zaniolo, C., "Database Language GEM", Proc. of ACM SIGMOD Conference on Management of Data, May 1983.