Modelling Information Preserving Databases: Consequences of the Concept of Time

Manfred R. Klopprogge
Peter C. Lockemann

Fakultät für Informatik, Universität Karlsruhe
Postfach 6380, D-7500 Karlsruhe

Abstract

Many modern database applications must preserve a record of the past over and above the current state of an application environment. For these applications, the concept of time is of central importance. Databases that model these applications cannot be based on the concept of state alone but must replace it by the concept of history as a function from a temporal domain to some value set. Such databases will be referred to as information preserving databases. The paper explores the consequences of the history concept mainly from a database design point of view. Firstly, the entity-relationship model is extended to include histories. Secondly - and this is the central topic - the paper introduces for it a framework for inferring states of the past that have not explicitly been stored in the database. The framework is based on the notion of uncertainty, and uses procedural means and ground rules for limiting uncertainty to a few well-defined situations. Thirdly, the paper reviews the update semantics which become slightly more complex than in traditional databases. An extensive example illustrates the various concepts developed in the paper.

1 Introduction

The classical database is a model of some real world system. At all times the contents of a database are intended to represent a snapshot of the state of an application environment [HM 81]. Such a database can best be characterized by the effect of its update operation: values in the database are **replaced** by new values. Furthermore, in answering a query the database management system (DBMS) makes an assumption of synchronism: the time difference between a change in the real world and the corresponding change in the database is so small as to be insignificant to the application. In other words, queries refer to the **present** state of the real world.

The notion of "present" is not without problems. Suppose a census is taken of some section of the population, recording a variety of demographic data. Then the present really is some fixed point in the past. Or consider a database for a satellite tracking station with delays in the range of seconds to hours, depending on satellite, between sending a signal and receiving the response. Such a database will only refer to the position of each satellite at some earlier albeit well-defined point in time. Hence there are different "presents" in the database depending on satellite.

Furthermore, there are growing numbers of applications that must deal with the past as well as with the present. Consider again our satellite tracking database from which one would like to infer the present position of a satellite by computing its orbit from a set of earlier positions. Other examples: clinical patient data for medical diagnoses, time series for statistical computations and trend analyses, successive measurements for machine control and diagnostics, ownerships of a gun for criminal investigations, deposits and withdrawals for checking accounts. In these applications, new data items must not replace old ones. Instead, the update operation **augments** (completes) the database thus **preserving** the older states of the database [Schu 77]. Clearly, in order to distinguish between various states of the real world, each item must receive a time-stamp.

All these different situations can be covered by the same concept: generalizing the traditional notion of state to the notion of **history**. Formally, a history is a mapping

$$h: \mathbf{T} \longrightarrow \mathbf{V}$$

from a set $\mathbf{T}$ of time representations to a set $\mathbf{V}$ of values.

As already indicated above, not all items within a database will undergo changes at the same time. Moreover, one might well imagine that for some items only the current value is of interest, whereas for others only some value in the past may be known, while still others require a history of the entire past. Consequently, a history should not apply to the database as a whole, rather each item should be allowed to have its own individual history.

What is to be considered an item? Since history is definitely an application-defined phenomenon, the concept of item should follow from the application. On the other hand, if a DBMS is to provide facilities for dealing with histories, the concept of item should follow some general rules. The obvious solution is to choose a semantic data model as a sort of "base model" and to extend it by mechanisms for histories. In this paper the entity-relationship model will serve as a base model [Che 76].

Traditional databases do not distinguish whether the real world phenomenon corresponding to a database item does currently exist, existed in the past or is expected to exist some time into the future. In an information preserving database these distinctions are the very essence of its function. In particular, if an item had an existence in the past but none at present, this fact will be preserved in its history. Non-existence will be expressed within this history by the "undefined value".

Consider now two distinctive and successive points in time within a history where the values are different. We may be interested in values in between. Hence, one of the mechanisms needed is the capability to infer states that have not explicitly been stored in the database. Take again as an example the satellite tracking database which may be used to predict the current position of a satellite, or a checking account database which may be used to derive the balance at all times. Hence, closely associated with each history will be a derivation function in case all states can be determined with certainity, and a - perhaps empty - set of approximation functions in cases where some uncertainty is left. In the extreme, we may even be unsure whether there was a defined value at all; we then describe this situation by an "uncertain value". Logical propositions on the database, as a consequence, cannot always be said to be true or false but must be considered to be "unknown". Hence, information preserving databases introduce a need for a ternary logic.

Surprisingly enough, too little attention has been paid in the literature to a systematic treatment of information preserving databases, even though many problems lend themselves very naturally to that approach, as shown above. The first one to have raised the issue of information preservation seems to have been Schueler [Schu 77]. On the other hand a number of authors have discussed the narrower theme 'time in databases' by introducing concepts like 'event' and 'process'. (see, e.g., [Fal 74, Bub 77, HM 78, FK 78, BFM 79, Bub 80, Bol 79, And 81, And 82, Ser 80, Bra 78]). These can nicely be used to model discrete and fully recorded real world behavior. However, most of the approaches fail when it comes to a full time perspective of data, i.e. when queries about the state of the world for any given instant rather than just for occurence times of events are to be answered. More general patterns of temporal change on the one hand and the treatment of incomplete and erroneous recording of histories on the other hand need to be explored.

An integral part of dynamics - and, hence, of history - is the concept of time. A number of discrete temporal systems (sometimes called calendar systems) can be found in the literature, e.g. [Bru 72, BFM 79, And 81, And 82]. From these one may conclude that a DBMS should avoid prescribing a single calendar system; rather it should offer facilities for defining the calendar system most suitable to the application.

This paper will concentrate on application-specific issues in information preserving DBMS, and will neglect implementation issues. Ch. 2 will be devoted to the extensions needed in the entity-relationship model. In particular, it will discuss how to include procedural elements in the model. Ch. 3 will introduce the mechanisms for dealing with uncertainty. Questions of database updates will be covered by ch. 4. A brief outline of a proposal for a schema definition language illustrated by an example can be found in ch. 5. For a more detailed discussion of the topic, the reader is referred to [Klo 83].

2 Extending the entity-relationship model

2.1 Basic elements of the entity-relationship model

We assume that the reader is familiar with the entity-relationship model (ERM). Hence, we restrict ourselves to the enumeration of the basic aspects of the model. The ERM distinguishes between two kinds of elements: structured elements called **entities** that are usually thought to model those objects of the real world that are of prime interest to the modeling process, and atomic elements called **values** that model properties that the counterparts of entities have in the real world. Values are associated with entities via **attributes**, i.e. a property is modelled as an attribute/value-pair. Two or more entities may enter into a **relationship** in which each entity plays a certain **role**. In addition, a relationship may be characterized by a set of values associated with it via attributes.

Correspondingly, an ERM schema consists of a set of **entity types** and a set of **relationship types**. Entity types are declared by name and a set of attribute/value_set-pairs. Relationship types are given by a name, a set of role/entity_type-pairs, and a set of attribute/value_set-pairs. Binary relationship types may be declared to represent functional dependencies: such a type may be 1:1, 1:N to N:1, where the functionality is true in the set of relationships of the type for each database state (in place of functional dependencies, more general cardinalities [ISO 82] could also be used). Further, a relationship type may be declared to be total in one or more of the associated entity types, meaning that all entities of each of the types currently in the database must participate in a relationship of the type considered.

In the remainder we shall sometimes refer to both an entity and a relationship as an **object**, to an attribute or a role as a **component**, and correspondingly to object types and component types.

2.2 Component histories

As pointed out in the introduction, histories should be associated with individual items in the database rather than the database as a whole. In the ERM, the most natural candidate for the item level is the component and not the object, as the following example will demonstrate.

Consider an entity type person. A person has properties that never change over its lifetime, such as birth_date and birth_place. The corresponding components have no history, that is, in the attribute/value-pair the value, once assigned, will never change. We shall call this a (temporally) **constant** component. Otherwise we refer to the component as (temporally) **variable**. Address, employer, last name (for female persons and - in "progressive" countries - for male persons) are examples of variable components. Variable components are represented by an attribute/value_history-pair or a role/entity_history-pair, where a value_history is a set of discrete time/value-pairs and an entity_history a set of discrete time/entity-pairs (or, technically more precise, time/entity_reference-pairs).

Values of constants or within pairs in histories may be undefined (**nil**), **uncertain** or **unknown**. Consider the property of social_security_no. A person is normally not assigned one until she or he leaves school. Once assigned, the number does not change any more. While intuitively one might consider social_security_no to be a constant, it is in this case a variable with a history consisting of two values, a value of undefined (**nil**) until the number has been assigned, and the number from the time of assignment on.

Hence, constant components are restricted to components that remain the same over the entire life span of the object. **nil** is a legitimate value for a constant component, e.g. date_of_first_childbirth for a (male) person. If a constant value is initially not known, the constant will be assigned **uncertain**, denoting the fact that the constant may have a **nil** or non-**nil** value. Once the value becomes known, it will replace **uncertain**.

The value of **unknown** may be considered a restriction on **uncertain**, meaning "uncertain but not nil", or "not yet known but definitely not nil".

Note, incidentally, that a representation with two pairs is sufficient for the social_security_no history. That the value at any arbitrary time may be computed, and how this is to be done (**nil** for all times before assignment, and the number for all times thereafter) must be expressed as an additional property of the component. We shall return to this point in chs. 2.4 and 3.1.

2.3 Object existence

In the preceding section reference was made to the life span of an object. In fact, it would have been more appropriate to refer to the life span of the real world counterpart, since the object is to be maintained in the database over the entire life span of the database. Consequently, there is indeed a situation where a history must be associated with the object as a whole, namely the existence of its real world counterpart. Technically, we solve the problem by augmenting the object by a mandatory **existence** attribute, with truth values as values. The existence component may again be constant or variable; in the latter case there exists an existence history.

More formally, the existence of an object is true during all times for which at least one of the remaining components has a defined value. Conversely, a constant component has the same invariant value only during those times for which the object existence is true, otherwise it is considered undefined. The existence of an object is false during all times for which all of the remaining components have the undefined value **nil**. Conversely, if a component history is not given over the entire time domain (is not a total function), its value is considered undefined during the time the object existence is false. It follows that the existence history of an object must be defined as a total function. Objects with a constant existence exist either at all times or never.

Similar to constant components, assignment of a truth value to the constant existence attribute may be deferred; in this case the value of **unknown** ("unknown whether true or false") is initially assigned.

We finally note the following restriction. Uncertainty will arise (aside from initializing constant components and existence) because history, while being a continuous phenomenon, is recorded only at discrete times. In order to define uncertainty we must first state what is certain. Hence we rule that recorded component histories must not contain **uncertain** as a value, and recorded existence histories must not contain **unknown** as a value.

2.4 Procedural aspects and general form of the schema

We noted before (ch. 2.2) that computing the non-recorded portions of a component history must follow rules that are idiosyncratic to that component. We also observed (ch. 1) that calendar systems may vary from application to application. Hence it will, in general, be unavoidable to include with each component its own procedures for computing non-recorded values. In turn, these procedures will have to rely on procedures defined on time, e.g., to determine into which interval between recorded times the desired time will fall, or to compute the date following a given date. Both problems may basically be solved by a mechanism akin to abstract data types. This mechanism may then be applied towards other value sets as well.

Consequently, an extended ERM schema is declared in the following steps.

(1) Declaration of component value sets other than the standard ones (such

as Boolean, real, integer). Each value set is defined in the form of a **structure** consisting of a name, a value set in the form {x ∈ B|P(x)} with B a base set and P(x) a predicate (if P(x) is missing, P(x) = true is assumed), a list of relations (Boolean functions) and a list of operations (functions with non-Boolean range).

(2) Declaration of histories.
Each history is also defined in the form of a structure and consists of a name, a time structure and a value structure (which were previously defined according to (1)), perhaps a predicate for further restricting the set of pairs forming a history, and a set of relations and operations. Note that the same history structure may be used in different components.

(3) Declaration of patterns
A pattern is a value or history structure together with at least one assertion, at most one derivation function, and zero, one or more approximation functions. Patterns are unique within the schema, i.e., they may be associated with exactly one object type. Assertions are formulated in order to enforce certain consistency constraints on update (see ch. 4.2).

(4) Declaration of objects.
Each entity type is introduced by a name followed by a list of components. A component is given by an attribute name (among them **existence**), by an indication whether the component is constant or variable, in case of a constant by the name of a value structure or value pattern, or in case of a variable by the name of a history structure or history pattern. For a relationship type, role components are given by role name, by an indication of their functionality and totality, by an indication of whether they are constant or variable, in case of a constant by a reference to an entity type, or in case of a variable by a history structure or history pattern where the value part is a reference to an entity type.

An extensive example that illustrates the form of a schema may be found in the appendix. The schema definition language, TERM, in discussed in detail in [Klo 81, Klo 83].

3 Dealing with Uncertainty

3.1 Derivation and Approximation

In the simplest case, queries to an information preserving database are of the kind [Klo 83] "which value (of some component) was effective in the real world at time t_q?"

(More complex query kinds are conceivable that take recording time into account; note, however, that these require a more extensive concept of history.)

The answer appears trivial if the component is constant or there is an explicitely recorded state for time t_q. Otherwise the system must try to compute a value for t_q from the recorded fragments of the history. If this can be done with certainty, we call the corresponding procedure a **derivation**, and each element in the history a **characteristic state** of the component. (More precisely, the characteristic states are just those elements that are needed to compute the states for all times t_q.) As a rule, whenever t_q is identical to some recorded t, the history value at t is chosen as an answer. Otherwise the derivation function is executed and its result is returned.

If the value at t_q cannot always be computed with certainty, we call the corresponding procedure an **approximation**. There may be a number of reasons: the times for which the history was recorded may be spaced too far apart (in the sense of derivation, the history may only represent a subset of the set of characteristic states), or the recorded values may themselves be inaccurate as in case of estimates or physical measurements. More than one approximation function may be supplied, e.g., both a linear interpolation and a least-squares method. As a rule, because of reduced confidence in the recorded values, the answer to the query will always be obtained by executing the specified approximation function.

If an approximation function is to be applied, it must explicitly been selected. Otherwise the derivation function is chosen by default. If no derivation functions exists, and no approximation function has been selected, the value is uncertain for all t_q for which there is no recorded state.

Three kinds of derivations or approximations are possible:
- component-local: computation is solely based on component history.
- object-local: computation is also based on other components of the same object (perhaps using their derivation or approximation functions).
- global: computation makes use of other objects as well.

Note, finally, that a derivation or approximation function may also be associated with a constant component, computing its (fixed) value. By necessity, the function is object-local or global.

3.2 Determining the object existence

What was said in ch. 3.1 holds for the existence attribute as well (although, obviously, only derivations are meaningful). Because existence is a total function (ch. 2.3), existence is either true or false, and uncertainty is expressed as "unknown whether true or false". In consequence of ch. 2.3, the following strategies are used in order to determine the existence value of an object.

Object existence plays a central role in determining the component values of an object (ch. 3.3). In addition, object existence may enter into assertions, global derivations and global approximations in the form of logical expressions. Consequently, the need to deal with uncertainty introduces a need for a ternary logic (ch. 3.4). However, once such a logic has been introduced there is no reason to restrict components with truth values to just the set {true,false}; rather we shall also permit the set {true,false,unknown}. We shall refer to the former value set as Boolean and to the latter as Kleenean.

3.3 Determining a component value

Again in accordance with ch. 2.3, we are now in a position to give a precise outline of the strategies for determining the values of an object component. We note that **nil** refers to the undefined value, whereas **uncertain** indicates that the value may either be an element of the value set considered, or **nil**. In particular, a truth value component may be determined to have an uncertain value meaning it could be one of **nil**, true, false and (in case of Kleenean) unknown.

a) Constant existence.
Note first that the value may have been initialized to unknown (ch. 2.3).

```
if existence_value ≠ unknown
then return recorded value
else if there exists at least one variable object_component
        with a non-empty history containing at least one
        value ≠ nil
     then return true
     else if derivation function is specified
          then return result of derivation function
          else return unknown.
```

b) Variable existence.
Note that the history may have been initialized to the empty set.

```
if an existence_value has been recorded for time t_q
then return existence_value for time t_q
else if there exists at least one variable object_component
        with a non-empty history containing a value ≠ nil for
        time t_q
     then return true
     else if derivation function is specified
          then return result of derivation function
          else return unknown.
```

a) Constant component.
Note that the value may have been initialized to **uncertain**.

```
case object_existence at t_q of
  true: if component_value ≠ nil
        then return recorded value
        else if derivation function is specified
             then return result of derivation function
             else return nil;
  false: return nil;
  unknown: return uncertain
end.
```

where object_existence at t_q is determined according to strategy a) or b) in ch. 3.2.

In case an approximation is requested, the strategy is instead

```
if object_existence at t_q ≠ false
then return result of approximation function
else return nil.
```

b) Variable component.
Note that the value may have been initialized to the empty set.

```
if a component_value ≠ nil has been recorded for time t_q
then return component_value for time t_q
else case object_existence at t_q of
       true: if derivation function is specified
             then return result of derivation function
             else return uncertain;
       false: return nil;
       unknown: return uncertain
     end.
```

The latter strategy is due to strategy a) or b) in ch. 3.2 which state that if a component value ≠ **nil** has been recorded for time t_q, then object_existence = true.

In case an approximation is requested, the strategy is the same as for a constant component.

3.4 Ternary Logic

As mentioned before, logical expressions may arise in the course of querying an information preserving database whose evaluation would have to follow the rules of ternary logic. Depending on the interpretation of the third truth value, a number of ternary logic calculi have been proposed [Res 69]. The one whose interpretation matches the one introduced above for unknown is due to Kleene. In this chapter we just list some basic properties and laws; for details the reader is referred to [Klo 83]. (The reader may also find a very general and comprehensive treatment of information incompleteness in databases in [Lip 79]).

Let I stand for unknown, T for true and F for false. Then the following table defines the Kleenean Logic.

p	¬p
T	F
I	I
F	T

q	p & q			p v q			p =>q			p <=> q		
q	T	I	F	T	I	F	T	I	F	T	I	F
T	T	I	F	T	T	T	T	I	F	T	I	F
I	I	I	F	T	I	I	T	I	I	I	I	I
F	F	F	F	T	I	F	T	T	T	F	I	T

Note that

$(p \Rightarrow q) \Leftrightarrow (\neg p \lor q)$
$(p \Leftrightarrow q) \Leftrightarrow ((p \Rightarrow q) \,\&\, (q \Rightarrow p))$

are tautologies as in the binary logic, but

$\neg(\neg a \Leftrightarrow a)$
$a \Rightarrow a$
$a \Leftrightarrow a$

are not. Neither are

$p \Leftrightarrow (p = \text{true})$
$(\neg p) \Leftrightarrow (p = \text{false})$

as may easily be checked by means of truth tables.

For the logical expressions mentioned in ch. 3.2, the following definitions are of importance. Let B be some base set. The extension of a predicate P, $\{x \in B | P(x)\}$ is defined as $\{x \in B | P(x) = \text{true}\}$. Consequently, $\{x \in B | \neg P(x)\} = \{x \in B | P(x) = \text{false}\}$. In general, $\{x \in B | P(x)\} \cup \{x \in B | P(x)\} \neq B$.

The quantifiers are defined as follows [Res 69]. Consider the cardinalities

$C_t = |\{x \in B | P(x) = \text{true}\}|$

$C_f = |\{x \in B | P(x) = \text{false}\}|$

$C_i = |\{x \in B | P(x) = \text{unknown}\}|$

$C = |B| = C_t + C_f + C_i$

	true for	false for	unknown for
$\forall x \in B: P(x)=\text{true}$	$C_t=C$	$C_f>0$	$C_f=0 \,\&\, C_i>0$
$\forall x \in B: P(x)=\text{false}$	$C_f=C$	$C_t>0$	$C_t=0 \,\&\, C_i>0$
$\forall x \in B: P(x)=\text{unknown}$	$C_i=C$	$C_i<C$	-
$\exists x \in B: P(x)=\text{true}$	$C_t>0$	$C_f=C$	$C_t=0 \,\&\, C_i>0$
$\exists x \in B: P(x)=\text{false}$	$C_f>0$	$C_t=C$	$C_f=0 \,\&\, C_i>0$
$\exists x \in B: P(x)=\text{unknown}$	$C_i>0$	$C_i=0$	-
$\exists_1 x \in B: P(x)=\text{true}$	$C_t=1 \,\&\, C_i=0$	$C_t>1 \lor C_f=C$	$C_t\leq 1 \,\&\, C_i>0$
$\exists_1 x \in B: P(x)=\text{false}$	$C_f=1 \,\&\, C_i=0$	$C_f>1 \lor C_t=C$	$C_f\leq 1 \,\&\, C_i>0$
$\exists_1 x \in B: P(x)=\text{unknown}$	$C_i=1$	$C_i\neq 1$	-

Notice the rules

$(\exists x \in B: P(x)=\text{true}) \Longleftrightarrow \neg(\forall x \in B: P(x)=\text{false})$
$(\exists x \in B: P(x)=\text{false}) \Longleftrightarrow \neg(\forall x \in B: P(x)=\text{true})$
$(\forall x \in B: P(x)=\text{true}) \Longleftrightarrow \neg(\exists x \in B: P(x)=\text{false})$
$(\forall x \in B: P(x)=\text{false}) \Longleftrightarrow \neg(\exists x \in B: P(x)=\text{true})$

but **not**

$(\exists x \in B: P(x)=\text{unknown}) \Longleftrightarrow \neg(\forall x \in B: \neg(P(x)=\text{unknown}))$
$(\exists x \in B: P(x)=\text{unknown}) \Longleftrightarrow (\forall x \in B: \neg(P(x)=\text{unknown}))$

Selection of elements from a set B is governed by the conventions

<u>some</u> z <u>from</u> $\{x \in B | P(x)\}$ defined as

```
   case (∃x∈B:P(x)=true) of
     true: z:= an arbitrary element of the set;
     false: z:= nil;
     unknown: z:= uncertain.
```

<u>that</u> z <u>from</u> $\{x \in B | P(x)\}$ defined as

```
   case (∃₁x∈B:P(x) = true) of
     true: z:= the unique element of the set;
     false: z:= nil;
     unknown: z:= uncertain.
```

The appendix gives numerous examples for predicates and set selections, almost all of them within function declarations. ($\forall$ is written as **all**, $\exists$ as **exists**, $\exists_1$ as **unique**, => as **impl**, $\{x \in B | P(x)\}$ as B **where** P(x); **this** refers to the currently considered element of the structure.)

4 Update semantics

4.1 Recording and correction

The traditional database cannot distinguish between an update that is due to a change of state in the real world, and an update that is caused by an improved perception of the same state. Not only is this distinction paramount to the proper functioning of an information preserving database, such a database is the only one in which the distinction is meaningful. Almost naturally, therefore, one will distinguish between two user roles with respect to update operations, namely: **recorder** and **referee**.

The recorder may install new objects, supplant an **uncertain** constant component, or add a new time/value-pair to a component history. In doing so, he must observe all consistency constraints in order to ensure that only plausible updates are performed. The referee is a specially authorized person, and knowledgeable enough to recognize inaccuracies or errors. He is permitted to change the value of a constant component or in a time/value-pair of the recorded history of a variable component. In particular, he may do so regardless of whether the consistency constraints are violated or not. The premise here is that a constraint mirrors an assumed law in the real world, and that the referee is in a position to determine whether the law needs some modification. We observe, though, that violating a constraint may have as a consequence that the constraint will never be satisfied during subsequent updates, hence a more discriminatory approach to the rights of a referee may actually be in order.

As in traditional databases, consistency will often be only maintained by a sequence of recordings. Hence the concept of transaction is essential to information preserving databases as well. This is mainly a matter of DML design, a topic we shall not go into any further in this paper.

4.2 Update constraints

There are three kinds of constraints that the DBMS must observe during recording.

(1) Test on set membership, if a base value set or a base history set is further restricted by predicate (ch. 2.4).

(2) Test whether assertions on patterns are satisfied (ch. 2.4). Two standard kinds of assertions are provided:
- **assertion key** specifies that the component to which the pattern refers has to have a unique value within the associated entity set or relationship set.
- **assertion false** indicates that an update of the component is never satisfied, i.e. the component is virtual. Consequently, the corresponding pattern must include a derivation and/or approximation function.

(3) Test on the implicit constraints expressing the rule that only values that are certain are recorded (ch. 2.3). These may now be formulated more precisely:

a) Constant existence.
old this.existence = unknown
    & **new this.existence** $\in$ {true,false}

b) Variable existence.
$\forall$ x $\in$ **new this.existence**: x.value $\in$ {true,false}

c) Constant component.
old this.attribute = **uncertain**
    & **new this**.attribute $\neq$ **uncertain**
    new this.attribute $\in$ component_value_set

d) Variable component.
(**old this**.attribute = {} v **new this**.attribute $\neq$ **nil**)
& **old this**.attriute $\neq$ **nil**
& **new this**.attribute $\neq$ **uncertain**
& $\forall$ x $\in$ **new this**.attribute:
        (x.time $\neq$ **nil** & x.time $\neq$ **uncertain** & x.value $\neq$ **uncertain**)
& **whole this**.attribute $\in$ history_value_set

Notation: **old** refers to the previously recorded component, **new** to the elements newly to be added, and **whole** to the result after update. **this**.attribute denotes the component value or history associated with attribute of the object of interest.

5 An example

The appendix contains an extensive example of a TERM (time extended ERM) schema which illustrates some of the foregoing concepts, and to which we already referred several times. The example has been taken from a banking application which typically must preserve a record of the past. The basic entity is the individual account. Transactions that cause changes to an account are also modelled as entities. Finally, because interest is credited to accounts, a third entity type, interest rate schedule, is added. Two relationship types relate an account to the rate schedule applying to it, and to the transactions affecting it. Changes to the account have to do for one with the transactions (deposits or withdrawals), for another with the interest accruing to it but which are credited to it only after each quarter of the year.

After all that hase been said in the paper so far the reader should have no difficulties in reading the example. Note that for the sake of completeness the entire date structure has been included; on first reading one may skip to the structures for the representation of histories.

6 Conclusions

One of the most interesting features of an information preserving database is the notion of uncertainty and the mechanisms for dealing with it. The paper could be viewed as a somewhat formalized approach to that subject, providing insights that to the authors' knowledge have hitherto not been reported. A second aspect of our work has been database design: extending the now classical entity-relationship model by additional concepts that may not only be useful for designing information preserving databases but allow sound decisions with regard to the structure of traditional databases. The schema definition language TERM, an extension of Pascal, was purposely developed as a programming language. As a third aspect of our work, this will permit the use of TERM not only as a design tool but also as an interface to information preserving DBMS. To determine the feasibility, a TERM compiler was implemented mapping the TERM interface to the interface of a network DBMS [Nun 82]. Fourthly, even if not used as an interface, translators could be built for such formalized schemas that mechanically generate network and relational interfaces with equivalent behaviour. Developing appropriate compilers and translators remains a topic for further research.

Bibliography

[And 81] T.L. Anderson: The Database Semantics of Time. Ph.D.thesis, Univ. Washington, 1981

[And 82] T.L. Anderson: Modeling Time at the Conceptual Level. Proc. 2nd Internatl. Conf. on Databases, Jerusalem, June 1982

[BFM 79] B. Breutmann, E. Falkenberg, R. Maurer: CSL: A Language for Defining Conceptual Schemas. In G. Bracchi, G.M. Nijssen (eds): Data Base Architecture, North-Holland 1979, 237-256

[Bol 79] A. Bolour: The Process Model of Data. Tech. Rep. 38, Lab. of Medical Info. Sci., Univ. California, San Francisco, 1979

[Bra 78] J. Bradley: Operations Data Base. Proc. 4th Internatl. Conf. on Very Large Databases, 1978, 164-176

[Bru 72] B. Bruce: A Model for Temporal Reference and its Application in a Question Answering Program. Artific. Intelligence 3 (1972), No. 1, 1-25

[Bub 77] J.A. Bubenko: The Temporal Dimension in Information Processing. In G.M. Nijssen (ed): Architecture and Models in Database Management, North-Holland 1977, 93-118

[Bub 80] J.A. Bubenko: Information Modeling in the Context of System Development. Information Processing 80, North-Holland 1980, 395-411

[Che 76] P.P.-S. Chen: The Entity-Relationship Model - Toward a Unified View of Data. ACM Trans. on Database Sys. 1 (1976), 9-36

[Fal 74] E. Falkenberg: Time-Handling in Database Management Systems. CIS-Rep. 07/74, Univ. Stuttgart 1974

[FK 78] A. Flory, J. Kouloumdjian: A Model for the Description of the Information System Dynamics. Lecture Notes on Comp. Sci. 65, Springer 1978, 307-318

[HM 78] M. Hammer, D.C. McLeod: The Semantic Data Model: A Modelling Mechanism for Data Base Applications. Proc. ACM SIGMOD Internatl. Conf. 1978, 26-36

[HM 81] M. Hammer, D.C. McLeod: Database Description with SDM: A Semantic Database Model. ACM Trans. on Database Sys. 6(1981), 351-386

[ISO 82] ISO TC 97/SC5/WG3 (J.J. v. Griethuysen, ed.): Concepts and Terminology for the Conceptual Schema and the

Information Base. Publ. No. ISO/TC97/SC5-N695, March 1982

[Klo 81] M.R. Klopprogge: TERM: An Approach to Include the Time Dimension in the Entity-Relationship Model. In P.P.-S. Chen (ed): Proc. 2nd Internatl. Conf. on Entity-Relationship Approach, ER Institute 1981, 477-512

[Klo 83] M.R. Klopprogge: Entity and Relationship Histories: A Concept for Describing and Managing Time Variant Information in Databases. Ph.D. thesis, Univ. Karlsruhe, 1983 (in German)

[Lip 79] W. Lipski jr.: On Semantic Issues Connected with Incomplete Information Databases. ACM Trans. on Database Sys. 4 (1979), 262-296

[Nun 82] H. Nunnenmann: Mapping TERM Schemas to UDS. Diploma thesis, Univ. Karlsruhe, Fak. Informatics, 1982 (in German)

[Res 69] N. Rescher: Many-Valued Logic. McGraw-Hill 1969

[Schu 77] B.-M. Schöler: Update Reconsidered. In G.M. Nijssen (ed): Architecture and Models in Database Management, North-Holland 1977, 149-164

[Ser 80] A. Sernadas: Temporal Aspects of Logical Procedure Definition. Information Sys. 5 (1980), 167-187

Appendix: TERM Schema of a Banking Application

```
define schema s_account;

%----- structures for the representation of times and values -------

structure
   st_icateg = integer;                          % represents interest categories

structure
   st_int_rate = real;                           % represents interest rates

structure
   st_account_nos = integer;                     % represents account numbers

structure
   st_name = packed array[1..20] of char;        % represents names

structure
   st_cur = real;                                % represents currencies

structure
   st_date =
      record d,m,y: integer end                  % represents calendar dates according
         where                                   % to the Gregorian calendar
            this.y >= 1582 and
            this.m >= 1 and this.m <= 12 and
            this.d >= 1 and this.d <= 31 and
            (this.d <> 31
              or this.m in [1, 3, 5, 7, 8,10,12]) and
            (this.d <> 30 or this.m <> 2) and
            (this.d <> 29 or this.m <> 2
              or this.y mod 4 = 0 and
                (this.y mod 100 <> 0
                  or this.y mod 400 = 0));

   relations
      function is_in_leap(t: st_date): boolean;  % does a date fall into a leap year?
      begin
         is_in_leap:= t.y mod 4 = 0 and
                      (t.y mod 100 <> 0
                       or t.y mod 400 = 0)
      end;
```

```
   function before_date(t1, t2: st_date): boolean;             % date t1 before date t2?
   begin
      before_date:= t1.y < t2.y
                   or t1.y=t2.y and t1.m<t2.m
                   or t1.y=t2.y and t1.m=t2.mand t1.d<t2.d
   end;

   function contemp_date(t1, t2: st_date): boolean;            % do t1 and t2 refer to the same day?
   begin
      contemp_date:= t1.d=t2.d and t1.m=t2.m and t1.y=t2.y
   end;

   function becon_date(t1, t2: st_date): boolean;              % t1 before t2 or t1 = t2?
   begin
      becon_date:= not before_date(t2, t1)
   end;

operations
   function ultimo_date(z: st_date): st_date;                  % last day of a month
   var ud: integer;
   begin
      if z.m in (1, 3, 5, 7, 8, 10, 12) then
         ud:= 31;
      else if z.m in (4, 6, 9, 11) then
         ud:= 30;
      else if z.m=2 and is_in_leap(z) then
         ud:= 29
      else if z.m=2 and not is_in_leap(z) then
         ud:= 28;
      ultimo_date.d:= ud;
      ultimo_date.m:= z.m;
      ultimo_date.y:= z.y
   end;

   function next_day_date(z: st_date): st_date;                % date of day following z
   var n: st_date;
   begin
      if z.d = 31 and z.m = 12 then begin                      % New Year's Eve
         n.d:= 1; n.m:= 1; n.y:= z.y + 1 end
      else if z = ultimo_date(z) then begin                    % last day of the month?
         n.d:= 1; n.m:= z.m + 1; n.y:= z.y end
      else
         n.d:= z.d + 1; n.m:= z.m; n.y:= z.y;
      next_day_date.d:= n.d;
      next_day_date.m:= n.m;
      next_day_date.y:= n.y;
   end;

   function prev_day_date(z: st_date): st_date;                % date of day preceding z
   var p: st_date;
   begin                                                       % this day has no predecessor,
      if z.d = 1 and z.m = 1 and z.y = 1582 then               % start of Gregorian calendar
         p:= nil
      else if z.d=1 and z.m=1 and z.y>1582 then begin          % New Year?
         p.d:= 31; p.m:= 12; p.y:= z.y - 1 end
      else if z.d = 1 and z.m > 1 then begin
         p.d:= 1; p.m:= z.m - 1; p.y:= z.y;
         p:= ultimo_date(p) end
      else begin                                               % not beginning of a month
         p.d:= z.d - 1; p.m:= z.m; p.y:= z.y end;
      prev_day_date.d:= p.d;
      prev_day_date.m:= p.m;
      prev_day_date.y:= p.y
   end;

   function least_recent_date(st: set of st_date;
                              z: st_date): st_date;            % max. date ≤ z in a set of dates
   begin
      least_recent_date:= that x from st where
                             all y from st where
                                becon_date(x, z) and
                                (before_date(y, x)
                                 or before_date(z, y))
   end;
```

```
%----- structures for the representation of histories -------------

structure
   hs_date_cur =
      history
         t: st_date;
         v: st_cur end;

structure
   hs_date_kleenean =
      history
         t: st_date;
         v: kleenean end;

structure
   hs_standard_exist =
      history
         t: st_date;
         v: kleenean end
      where                                                     % there is at most one time
         all s1, s2 from this where                            % interval during which the
         ((s1.v = true and s2.v = false and before_date(s1,s2)) % existence is true
         impl all s from this where                            % (this models the life of
                 before_date(s2, s) impl s.v = false)          % living beings etc.)
         and
         ((s1.v = false and s2.v = true and before_date(s1,s2))
         impl all s from this where
                 before_date(s, s1) impl s.v = false)
         and
         ((s1.v = true and s2.v = true and before_date(s1,s2))
         impl all s from this where
                 before_date(s1, s) and before_date(s, s2)
                    impl s.v = true;

   operations
      function start_ex_standard(h:hs_standard_exist):st_date;   % beginning of existence
      var s: state of hs_standard_exist;
      begin
         s:= that x from h where                                 % oldest known state with s.v = true
               all y from h where
                    not y.v or becon(x.t, y.t);
         if s <> nil then                                        % is there one
            start_ex_standard:= s.t
         else
            start_ex_standard:= uncertain
      end;

      function d_standard(h: hs_standard_exist;                  % auxiliary function for
                          z: st_date): kleenean;                 % pattern derivation
      var ds: state of hs_standard_exist;
      begin
         ds:= that s1 from h where
              s1.t=least_recent_date(those tx from st_date where
                                     exists s2 from h where
                                        s2.t = tx, z);
         if ds <> nil then                                       % function is total
            d_standard:= ds.v
         else
            d_standard:= unknown
      end;

structure
   hs_int_rate =
      history
         t: st_date;
         v: st_int_rate end;

   operations
      function d_int_rate(h:hs_int_rate;                         % auxiliary function
                          z:st_date): st_int_rate;               % derivation
      var dz: state of hs_int_rate
      begin
         dz:= that s1 from h where
              s1.t=least_recent_date(those tx from st_date where
                                     exists s2 from h where      % all points in time of h
                                        s2.t = tx, z);
         if dz <> nil then                                       % function is total
            d_int_rate:= dz.v
         else
            d_int_rate:= uncertain
      end;
```

```
%----- patterns for the components of 'rate_schedule' --------------

pattern
   int_exist = hs_standard_exist;

   derivation
      function deriv_int(pz:↑rate_schedule; z:st_date):kleenean;
      begin                                                    % total because of d_standard
         deriv_int:= d_standard(pz↑.existence, z)
      end;

pattern
   int_icateg = st_icateg;

   assertion
      key;

pattern
   int_debit = hs_int_rate;

   derivation
      function deriv_debit_rate(pz:↑rate_schedule;
                                 z:st_date): st_int_rate;
      begin
         deriv_debit_rate:= d_int_rate(pz↑.debit_rate, z)      % total because of d_int_rate
      end;

pattern
   int_credit = hs_int_rate;

   derivation
      function deriv_credit_rate(pz:↑rate_schedule;
                                  z:st_date): st_int_rate;
      begin                                                    % total because of d_int_rate
         deriv_credit_rate:= d_int_rate(pz↑.credit_rate, z)
      end;


%----- patterns for the components of 'account' --------------------

pattern
   account_exist = hs_standard_exist;

   derivation
      function deriv_acc_ex(pa:↑account;
                             z:st_date): kleenean;
      begin
         deriv_acc_ex:= d_standard(pa↑.existence, z)
      end;

pattern
   account_no = st_account_nos;

   assertion
      key;


pattern
   account_icateg =
      history
         v: st_date;
         t: st_icateg end;

   derivation
      function deriv_icateg(h:account_icateg;
                             z:st_date):st_icateg;
      var dz: state of account_icateg;
      begin                                                    % derivation is total
         dz:= that s1 from h where
              s1.t=least_recent_date(those tx from st_date where
                                     exists s2 from h where
                                        s2.t = tx, z);
         if dz <> nil
            deriv_icateg:= dz.v
         else
            deriv_icateg:= uncertain
         end;
```

```
pattern
   account_balance = hs_date_cur;                                      % for virtual component

   assertion
      false;

   derivation                                                          % current balance
      function deriv_account(pa:↑account;z:st_date):st_cur;
      var ds: st_cur; zi: st_date;
      begin
         zi:= start_ex_standard(pa↑.existence);                        % day of opening an account
         ds:= 0;
         while becon_date(zi,z) do begin                               % sum of all daily balances
            ds:= ds + pa↑.day_balance at zi;
            if zi=ultimo_date(zi) and zi.m mod 3=0                     % end of quarter?
            then
               ds:= ds + pa↑.noncred_interest at zi;                   % interest credited
               zi:= next_day_date(zi) end;
         deriv_account:= ds end                                        % defined as total
      end;

pattern
   account_dbal = hs_date_cur;

   assertion
      false;

   derivation                                                          % balance for the day

      function deriv_dbal(pa:↑account; z:st_date): st_cur;
      var dt: st_cur; sta: set of transaction;
          pt: ↑transaction;
      begin
         dt:= 0;
         sta:= those tr from transaction where                         % all transactions on
                  tr.existence at z and                                % an account for day z
                  exists at from acc_ta where
                     at.acc = pa and at.ta = tr;
         while sta <> () do begin                                      % total amount of all
            pt:= some s from sta;                                      % transactions
            sta:= sta without (pt);
            dt:= dt + pt↑.amount end;
         deriv_dbal:= dt end
      end;

pattern
   account_noncred_interest = hs_date_cur;

   assertion                                                           % for virtual attribute
      false;

   derivation                                                          % interest accrued from
      function deriv_noncred_int(pa:↑account; z:st_date):st_cur;       % current quarter that

      var doz, db: st_cur; zi: st_date;                                % have not been credited
          z: ↑rate_schedule; zf: st_int_rate;                          % yet
      begin
         zi.d:= 1; zi.y:= z.y;
         zi.m:= ((z.m - 1) div 3) * 3 + 1;                             % zi:= begin of quarter
         if not pa↑.existence at zi then
            zi:= start_ex_standard(pa↑.existence);                     % day account opening
         doz:= 0;
         while becon_date(zi,z) do begin
            db:= pa↑.day_balance at zi;
            z:= pa↑.rs at zi;                                          % rate schedule appli-

            if db < 0 then                                             % cable to account on
               zf:= z.debit_rate                                       % day zi
            else
               zf:= z.credit_rate;
            doz:= doz + db*zf/3600 end;                                % sum of all daily interests

         deriv_noncred_int:= doz end
      end;
```

```
%----- patterns for the components of 'transaction' ---------------

pattern
   transaction_exist =
      history
         t: st_date;
         v: kleenean end
      where
         exists s from this where s.v                               % each history has at

            impl unique s from this where s.v;                      % most one state with

                                                                    % s.v=true (point event)

   derivation
      function deriv_ta(pt: ↑transaction;
                        z: st_date): kleenean;
      var x: state of transaction_exist;
      begin
         x:= that s from pt↑.existence
               where s.t = z;
         if x <> nil then
            deriv_ta:= x.v
         else if exists s from pt↑.existence where s.v then
            deriv_ta:= false
         else
            deriv_ta:= unknown
      end;

%----- patterns for the components of 'rs_acc' --------------------

pattern
   rs_acc_rs =
      history
         t: st_date;
         v: ↑rate_schedule end;

   assertion                                                        % for virtual role
      false;

   derivation
      function deriv_rs(pr:↑rs_acc; z:st_date):↑rate_schedule;
      begin
         deriv_rs:= that z from rate_schedule where
                        z.int_category=pr↑.acct↑.int_category at z
      end;
```

```
%----- entity types and relationship types ----------------------

   entity type
      rate_schedule;
      existence
         variable int_exist;
      attributes
         int_category constant int_icateg;
         debit_rate variable int_debit;
         credit_rate variable int_credit;

   entity type
      account;
      existence
         variable account_exist;
      attributes
         no constant account_no;
         owner constant st_name;
         int_category variable account_icateg;                  % interest category
         balance variable account_balance;                      % current balance
         day_balance variable account_dbal;                     % total for each day
         noncred_interest variable account_noncred_interest;    % not yet credited interest

   entity type
      transaction;
      existence
         variable transaction_exist;
      attributes
         amount constant st_cur;

   relationship type
      rs_acc;
      existence
         constant kleenean;
      roles
         rs one variable rs_acc_rs;                             % current rate schedule 'rs'
         acc total constant ↑account;                           % for account 'acc'

   relationship type
      acc_ta;
      existence
         constant kleenean;
      roles
         acc one constant ↑account;                             % transaction 'ta' belongs
         ta total constant ↑transaction;                        % to account 'acc'
```