20

Modeling and Implementation of Temporal Aspects of Objects in Commercial Information Systems

Volker Gruhn
University of Dortmund, Computer Science
D-44221 Dortmund
email: gruhn@ls10.informatik.uni-dortmund.de

Martin Fegert, Monika Schneider
o.tel.o telecommunications GmbH
Universitätsstraße 140, D-44799 Bochum
email: [fegert,mschneid]@lion.de

Abstract

In this article, we discuss how temporal aspects of information systems can be modeled and implemented. We introduce extensions of an entity–relationship model which allow us to model temporal aspects. Moreover, we describe how these extended entity-relationship models are used to generate an application database which supports objects with temporarily restricted validity. We support historical objects (i.e. objects which are not valid any longer) and future objects (i.e. objects which will be valid in the future). Our approach is driven by requirements from a real-world project. In this project we had to support temporal aspects of objects in an information system supporting business processes from the area of housing construction and administration.

Keywords

temporal database, object, future object, historical object, TEER

1 INTRODUCTION

In many commercial information systems, the temporal aspects of information are only considered by deleting information. Sometimes rudimentary functions for versioning and archiving of old information are available. In other cases, information either is valid (i.e. it has to be stored) or it is not valid any longer (i.e. it can be deleted). This simple mechanism of deletion sometimes oversimplifies the notion of *validity* of information. For certain kinds of information,

Data Mining and Reverse Engineering S. Spaccapietra & F. Maryanski (Eds.)
© 1998 IFIP. Published by Chapman & Hall

it is not appropriate just to administer the currently valid values. It may also be interesting to know how this information looked like before it was updated or to store that this information will be updated at some time in the future. This applies most of all to information that naturally changes from time to time (e.g. addresses of persons, bank accounts, salary).

For documents the recording of modifications and the archiving of earlier versions are well-known and widespread (G.Kappel, Pröll, Rausch-Schott and Retschitzegger 1995). Once a report has been released or a letter has been sent, the report or letter will normally not be changed without recording the former version. Modifications of documents can be tracked and versions of documents can be restored. If clearly defined change management processes are in place, it will be enforced to record why the modification has been done, who did it and to whom the new version has been distributed. If a new version is sent to a recipient of the earlier version, a change history section will be added in order to allow a guided reading of the new version. Document management systems usually provide comfortable support for these clearly structured processes.

Surprisingly, the situation is completely different as soon as we switch to structured pieces of information stored in information systems. Generally speaking, in most commercial information systems information has a limited lifespan (between creation and deletion). Once this lifespan is over, information disappears completely. The need to manage the complete lifetime of information (instead of deleting information at the end of its lifespan) has been recognized, e.g. in insurance companies, banks, housing construction and administration companies. Only a very few approaches to the efficient implementation of information systems supporting temporal aspects of objects are available (Ait-Braham, Theodoulidis and Karvelis 1994). Generally speaking, the temporal aspects of information are subject to research in the area of temporal databases (Clifford and Crocker 1993, Özsoyoglu and Snodgrass 1995, Jensen, Clifford, Gadia, Segev and Snodgrass 1994, Tansel, Clifford, Gadia, Jajodia, Segev and Snodgrass 1993, Pissinou, Snodgrass, Elmasri, Mumick, Ozsu, Percini, Segev and Theodoulidis 1994). Our approach eases management of temporal aspects by simple modeling constructs and automatic generation of database structures and access functions.

In this article, we describe an approach which supports temporal aspects of information throughout information system development and usage. We do not only propose mechanisms for managing temporal aspects from a database point of view, but in addition we support the modeling of temporal aspects on the conceptual data modeling level.

Our experiences with temporal aspects of objects are related to business process oriented software development (Gruhn and Wolf 1995) in the area of housing construction and administration. We have developed an information system which supports all business processes from the area of housing construction and administration for a consortium of eight customers. We spent

about 160 person years in developing this system. Questions like *Which tenant lived at which address?, Which tenant had which bank account at a certain time?* turned up in various business processes. They underpin the need for history recording. To deal with a tenant changing his bank account at the end of the current month is a typical example showing that future modifications have to be managed.

In brief, the goal of our approach is as follows: we support comfortable modeling of temporal aspects of objects on the conceptual level. Based on this conceptual model, we generate a database schema and we provide an application programming interface which offers functions for access to objects and the administration of their histories and futures. We identify in the conceptual model objects of which types will be subject to historical recording and/or future modifications. For objects of these types we ensure that modifications will be recorded and that future modifications are applied automatically at the specified time. Our approach has been implemented in the business process management environment $\mathcal{L}eu$ (Dinkhoff, Gruhn, Saalmann and Zielonka 1994, Gruhn, Pahl and Wever 1995).

Section 2 discusses how temporal aspects can be modeled in the context of entity-relationship modeling. Section 3 introduces a concept about how to handle historical and future objects. The implementation of this concept on top of a relational database management system is discussed in Section 4. We do not discuss in detail why we used a relational database management system (and not an object-oriented one), because the requirement of using the systems to be developed at several customers entailed the use of a relational system. In other words, the information systems to be developed should be deployable on a broad commercial base. Finally, Section 5 sums up our experiences and concludes with an overview about the research work still to be done.

2 ENTITY-RELATIONSHIP BASED MODELING OF TEMPORAL ASPECTS

Object modeling in $\mathcal{L}eu$ is based on extended entity-relationship diagrams (EER diagrams) (Dinkhoff et al. 1994). In this section we take a closer look at a simplified object model from the area of apartment administration. Figure 1 shows this model called *lease management*. For reasons of simplicity, this example does not contain multivalued attributes, composite attributes and constraints, even though the extended entity-relationship diagrams support them.

Relationships can be of the cardinalities 1:1, 1:n, or n:m. This is depicted in single-ending and multi-ending edges. Furthermore, they can be mandatory or optional. This is represented by solid and dashed lines. We do only support binary relationships without attributes. This model, despite very simple, turned out be powerful enough to develop a variety of information systems.

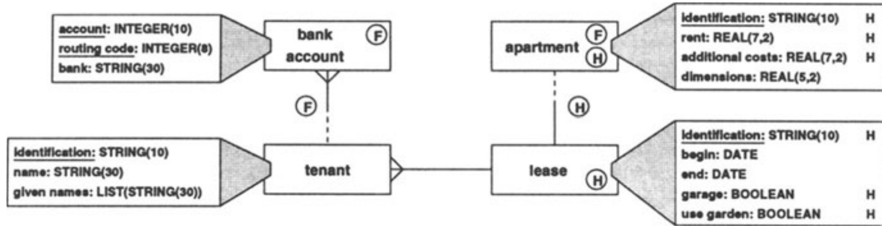

Figure 1 Object model for lease management

Some object-oriented features have been introduced into the entity-relationship modeling as supported by $\mathcal{L}eu$. First of all, we consider entity types as object types. Each entity has a unique object identity. We use the term *object* to denote an entity. Objects can be of a predefined format (Postscript, WordPerfect documents, etc.) or of a complex type. Objects of these complex types are described by means of their attributes. Attributes can be mandatory or optional. Mandatory attributes can be used to compose a user-defined primary key of an object type. In Figure 1 the primary keys are underlined. Non-primitive attribute types are either lists that can be composed of primitive types or they are enumeration types which can be defined by the user.

Encapsulation of object types by means of user-defined operations is another object-oriented feature of $\mathcal{L}eu$ object modeling. Each object can only be manipulated by operations attached to the corresponding object type. Moreover, we use the concept of delegation between object types (Wieringa, de Jonge and Spruit 1994) in order to define relationships between closely related object types.

In order to capture the temporal semantics of objects the object model of $\mathcal{L}eu$ is extended by features of the Bitemporal Conceptual Data Model (BCDM) (Jensen, Soo and Snodgrass 1989) and the Temporal EER Model (TEER) (Tansel et al. 1993). By marking an attribute with a **H** we define that it is subject to historical recording. Thus, an object type with at least one marked attribute is subject to historical recording. The history of relationships is automatically recorded if both of the related object types are marked. If only one object is marked it does not make sense to record the relationship, because it does not have a target in the historical database.

By marking an object type with a **F** at modeling time, we define that future modifications of existing objects can be stored. Then, at runtime t_0 it may be known that an object will change its value at a future time $t_1, t_1 > t_0$. In order to ensure that the modification is not occasionally forgotten at time t_1, the future modification can be described at time t_0 and is automatically started at t_1. Whether or not the modification can successfully be finished at time t_1, depends on the database state at t_1 and cannot be decided at t_0. The *future* of an object records all future modifications of this object*. The

*Even though not future objects, but future modifications are recorded, we use the term

same applies to relationships. In this way the whole lifespan of objects and relationships can be recorded.

There is a tradeoff between storage of as much as possible historical objects and future modifications on the one hand and performance and data volume on the other hand. The storage of earlier objects as well as the storage of future objects costs database space and it reduces performance, e.g. objects do not have to be simply deleted, but their lifespan values have to be updated. Since the $\mathcal{L}eu$ approach allows to explicitly model which object types are subject to historical recording and future modification recording, it allows user-defined compromises between complete recording and performance and data volume efficiency. This is based on letting the user decide objects of which types have to be recorded (in contrast to recording each modification of each object).

3 A CONCEPT FOR DEALING WITH TEMPORAL ASPECTS

We support the modeling of temporal aspects of object types on the level of conceptual models. This is done by marking object types with an **H** or with an **F**, thus indicating that they are subject to historical recording or that future modifications can be stored automatically. Thus, conceptual modeling is relieved from describing details about how historical and future information is stored. The definitions given in the conceptual model are the base for supporting the complete lifespan of objects. The database schema generated from the conceptual model (Gruhn et al. 1995) allows to distinguish between different versions of objects and to determine their lifespans.

The basic idea for the complete lifespan management is the distinction of different versions of objects. Either versions are valid, they were valid or they will be valid. All currently valid objects are members of the set of current objects stored in the application database **ADB***. All former versions of objects are members of the set of historical objects stored in the historical database **HDB**. Finally, all objects with future validity are members of the set of future objects stored in the future database **FDB**.

In contrast to other approaches (Clifford and Crocker 1993, Gadia 1988, Tansel et al. 1993) in which all objects (historical, current, future) are administrated as one set of objects, we distinguish between the three sets of objects mentioned. The benefit is that the access to current objects is as fast as if we were not dealing with temporal aspects at all. The additional sets of historical and future objects are only accessed when temporal aspects have to be checked. This approach - even though less elegant than a seamless integration of the three sets of objects - is efficient and does not burden the access to actual objects by administration overhead for temporal aspects.

---

future object in the following. It denotes objects which may be created by future modifications.

*The notion of database refers to a logical database. Technically speaking, all databases discussed in the following are components of only one physical database.

Section 3.1 refers to other approaches for dealing with temporal aspects of information and basic results of the research on temporal databases. In Section 3.2 we discuss the notation used to describe temporal aspects of objects. Then, Section 3.3 discusses what the generated databases for historical, current and future objects look like. Sections 3.4, 3.5, and 3.6 describe the concepts for dealing with historical objects, future objects and objects which are subject to historical and future recording.

3.1 Related Work

Our implementation focuses on an implementation on top of a relational database management system, because the use of such a system was a must-requirement for the system to be developed. The handling of temporal aspects has been discussed in the context of relational databases and query languages (Hou and Özsoyoglu 1993, Clifford, Croker and Tuzhilin 1994, Ait-Braham et al. 1994, Snodgrass 1995). A *temporal database* is a database supporting some aspect of time (Jensen et al. 1994). Objects or attribute values of a temporal database are associated with *timestamps.* A timestamp may be a *transaction time* which is the time when an object is physically stored in the database. A timestamp may also be a *valid time* which is the time when an object starts to exist in the modeled reality (Snodgrass and Ahn 1985, Snodgrass and Ahn 1986). Relations containing one of these timestamps are called *transaction-time relations* and *valid-time relations* respectively. Relations with both timestamps are called *bitemporal relations.*

When dealing with temporal databases, one of the most important tasks is the definition of an appropriate data type for these timestamps. In (Gadia and Vaishnav 1985, Gadia 1988) a *temporal element* is introduced as a finite union of intervals. Such a temporal element defines the *lifespan* (Clifford and Crocker 1993) of the object it is associated with.

Our application database contains *snapshot relations* (Jensen et al. 1994) without any timestamps. Its objects are currently valid. Our historical database consists of transaction-time relations. The objects stored in these relations are associated with a transaction time. Therefore, they are *temporally homogeneous.* That means the lifespans of all attribute values within a historical object are identical. Finally, our future database contains bitemporal relations. A future object is associated with a transaction time showing its definition time and a future valid time.

In most approaches a temporal element is implemented as one attribute of a temporal relation storing intervals or as two attributes storing a start- and an end-time (Tansel et al. 1993). In our approach we only store start-times. Since we have a zero-information-loss model (i.e. no information about an object's history is deleted) (Bhargava and Gadia 1993) and the lifespans of our objects cannot have gaps, the end-time of a version of an object equals

the start-time of the next version of this object minus one chronon. A *chronon* (Jensen et al. 1994) is the shortest duration of time supported by a temporal database.

3.2 Notations

We use the relational model to define our database schema and the relational algebra to define the manipulations of objects and relations. The relational table* representing an object type is called OT and the table for a relationship is called RT. The index A denotes tables of the ADB. For example, the table OT_A represents a current object type table. Analogously, the indexes H and F denote tables of the HDB and FDB. The attributes of the object types are denoted as $A_1, \ldots, A_n$. Further columns of the tables are the surrogates S, S_F and S'. S and S_F are surrogates being primary keys of a current or a future table. S' is a column containing also surrogates, but they need not be unique. They are used in historical and future tables where more than one version of an object or a relation may be stored. Further columns of the tables are $R_1, \ldots, R_m$. They represent foreign keys storing relations surrogates of related objects. The tables of the HDB and FDB additionally contain columns called T and V storing timestamps for versions of objects and relations and a column called M storing the kind of manipulation (insert, update, delete, etc.). Furthermore, we use the ω as null-value.

3.3 Generating database schemas

Object models are the base for generating an information system compare Figure 2). We use the object model to generate a relational database schema. The tables of this schema store all objects and relations produced, manipulated, and retrieved during the execution of business processes with $\mathcal{L}eu$. As discussed in (Gruhn et al. 1995), the internal split of objects over several tables remains invisible, because it is encapsulated by the database access functions. The database schema is divided into three schemas representing the sets of historical, current, and future objects and relations. They are related by a process executing future actions and a mechanism saving versions of manipulated objects and relations in the HDB. In the following the database schemas are described.

ADB: The application database is built up as follows: for each object type at least one table is generated. The relational schema of this table is defined as

$$OT_A = (S, A_1, \ldots, A_n, R_1, \ldots, R_m) \quad (1)$$

*For simplicity we only use the term 'table' in the following.

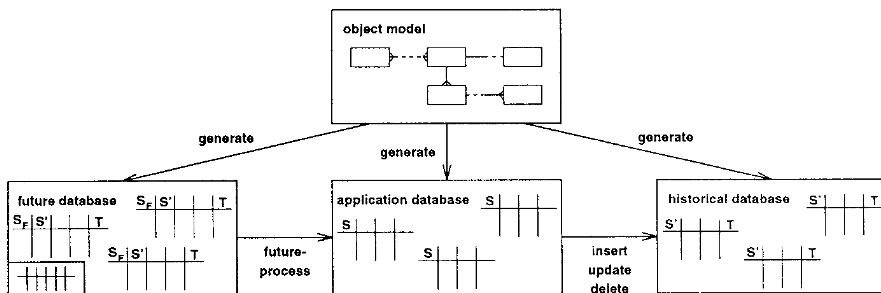

Figure 2 Generating database schemas

S is called *surrogate* and represents the primary key of the schema. Furthermore, the table contains one column for each of the attributes $A_1, \ldots, A_n$. The columns $R_1, \ldots, R_m$ represent 1:1- or n:1-relationships. They are used to store foreign keys of related objects.

For each n:m relationship a table is generated. It contains two foreign keys for the surrogates of the two related object types. The schema of this table is defined as

$$RT_A = (R_1, R_2) \tag{2}$$

As an example, Figure 3 shows the current table of the object type *apartment* (see Figure 1) with its 1:1-relationship to the object type *lease*. The table already contains two objects with relations.

$apartment_A = (S, identification, rent, additional\ costs, dimensions, R_lease) =$

S	identification	rent	additional costs	dimensions	R_lease
abc	S_25	580.00	95.00	56.40	xyz
def	X_83	1137.00	129.00	84.38	uvw

Figure 3 Relational Table in the ADB

HDB: In the historical database the structures of the generated tables are similar to the structures of the current tables:

$$OT_H = (S', A_1, \ldots, A_l, R_1, \ldots, R_k, T_H, M) \tag{3}$$

One difference is that the tables only contain columns for those attributes $A_1, \ldots, A_l$ and relationships $R_1, \ldots, R_k$ which are subject to historical recording. In addition, each table contains two further columns. One contains the transaction time T_H which is the commit time of the transaction which manipulated the object or the relation. Because in the history the valid time is

the same as the transaction time we store only one timestamp for historical objects and relations. Column M stores the kind of the last action manipulating the object (refer to Section 4.4). The surrogate column S' is not used as primary key, because its values are not unique as soon as more versions of an object with different timestamps are stored. The combination of surrogate and transaction time is the primary key in the historical database. The relational schema of a historical n:m-relationship is defined as

$$RT_H = (R_1, R_2, T_H, M) \quad (4)$$

As an example, Figure 4 shows the table of the historical *apartment* objects and their relations.

$apartment_H = (S', identification, rent, additional\,costs, R_lease, T_H, M) =$

S'	identification	rent	additional costs	R_lease	T_H	M
abc	S_25	510.00	73.00	ω	Jan-1-92:9:04AM	Insert
abc	S_25	510.00	73.00	rst	Jan-1-92:9:10AM	R_Insert
abc	S_25	540.00	81.00	rst	July-1-95:4:43PM	Update
abc	S_25	540.00	81.00	ω	Feb-1-96:0:21AM	R_Delete
abc	S_25	580.00	95.00	ω	Feb-1-96:0:39AM	Update
abc	S_25	580.00	95.00	xyz	Feb-1-96:0:45AM	R_Insert
def	X_83	1137.00	129.00	ω	Aug-8-95:2:08PM	Insert
def	X_83	1137.00	129.00	uvw	Nov-1-95:8:13AM	R_Insert

Figure 4 Relational Table in the HDB

FDB: The structure of the future database schema differs a little from the other schemas. It consists of tables for all object types and relationships which are subject to future modifications:

$$OT_F = (S_F, S', A_1, \ldots, A_n, B) \quad (5)$$

$$RT_F = (S_F, R_1, R_2, B) \quad (6)$$

The future tables for object types contain two surrogate columns. One of them is a future surrogate S_F which is used as primary key for a future object. The other surrogate S' is the actual surrogate which identifies an object in the application database. Because a future table may store different versions of an object its surrogate cannot be used as primary key in the future database.

For each relationship which is subject to future modifications a table is generated — even for the 1:1- and 1:n-relationships. This is necessary when the object type the relationship is normally included in does not have a future table of its own. These tables also contain a column S_F for a future surrogate.

All future tables contain one additional column B with a foreign key referencing the surrogate S_{AT} of the so called *action table* which is defined as

$$AT_F = (S_{AT}, T_F, V_F, TR, SN, M) \tag{7}$$

This table stores all manipulating actions M which should be carried out on the objects and relations in the future and the timestamps of these actions. One timestamp is the transaction time T_F which is the commit time of the transaction creating a future action. The other is the valid time V_F representing the point of time at which a future action should be executed and therefore a new version of an object or a relation will be valid. Furthermore, the action table contains columns for a transaction TR and a sequence number SN. The purpose of the column TR is to group certain manipulating actions into transactions, which are translated into actual database transactions when they are executed. The sequence number defines the order in which the actions are executed within the transactions. For more details about the semantics of the action table we refer to the Sections 3.5 and 4.2.

As an example, Figure 5 shows the future tables of the object type *apartment* and its relation. The action table AT_F contains two transactions. The first consists of three future actions. They will change the rent and the additional costs of the apartment *X_83* ($S' = def$), delete the relation to the old lease, and create a new relation to another lease. The second transaction will change the additional costs of the apartment *S_25* ($S' = abc$).

$apartment_F = (S_F, S', identification, rent, additional\ costs, dimensions, B) =$

S_F	S'	identification	rent	additional costs	dimensions	B
a1b	def	ω	1192.00	137.00	ω	b1
j7e	abc	ω	ω	98.00	ω	b4

$apartment - lease_F = (S_F, R_apartment, R_lease, B) =$

S_F	R_apartment	R_lease	B
l0s	def	uvw	b2
k5q	def	opq	b3

$AT_F = (S_{AT}, T_F, V_F, TR, SN, M) =$

S_{AT}	T_F	V_F	TR	SN	M
b1	Feb-10-96:10:57AM	Jan-1-97:0:00AM	1	1	Update
b2	Feb-15-96:8:32AM	Jan-1-97:0:00AM	1	2	R_Delete
b3	Feb-15-96:8:40AM	Jan-1-97:0:00AM	1	3	R_Insert
b4	April-3-96:6:02PM	Jan-1-97:0:00AM	2	1	Update

Figure 5 Relational Tables in the FDB

3.4 History of Objects

The historical database HDB stores all historical objects and relations. Whenever an object is inserted, updated or deleted or a relation is inserted or deleted, the action, the concerned object/relation, and the timestamp is stored in the HDB. If, for example, an object is updated, the following object is inserted in the historical table of the object type:

$$ot_H = \pi_{S,A_1,\ldots,A_l,R_1,\ldots,R_k}(\sigma_{S=s}(OT_A)) \bullet t_H \bullet \text{'update'} \quad (8)$$

First, the operation $\sigma_{S=s}$ selects the object with the surrogate s from the current table OT_A. Then, the operation $\pi_{S,A_1,\ldots,A_l,\ldots,A_l,R_1,\ldots,R_k}$ projects the columns for the surrogate, the history attributes, and the history relations. The resulting object is combined with the transaction time t_H which is the commit time of the update operation, and the kind of the manipulation which is *update*.

An significant element of dealing with historical databases is the information retrieval. For retrieving a historical object or relation at a given time in the past we define a function

$$f_H \; : \; TIME \times \{BOOLEXP\} \times HDB \longrightarrow HDB \quad \text{with}$$

$$f_H(t, \varphi, R_H) = \sigma_{M \neq' delete'}(\sigma_{T_H = max(T_H)}(\sigma_{T_H \leq t \wedge \varphi}(R_H))) \quad (9)$$

The parameters of this function are a transaction time t, a formula φ (consisting of logical operations, e.g.: $S' = s$) which can be evaluated to *true* or *false* and a historical table R_H. The function first selects all objects of the historical table which fulfil the formula and whose transaction time is equal to or less than the given time. As next step the object with the highest transaction time is selected from the result of the first selection. If this is a *delete* operation, then f_H intentionally delivers the empty set, because the object did not exist anymore at the selected time. In all other cases f_H delivers the value of the object valid at the selected time.

Now we can use the function to retrieve a table containing the historical object ot_H with surrogate s at a given historical time t_H:

$$ot_H = \pi_{A_1,\ldots,A_l}(f_H(t_H, S' = s, OT_H)) \quad (10)$$

For example, the historical apartment with surrogate *abc* at the time *Nov-15-95:0:00AM* is determined by

$$\pi_{identification,rent,additional costs}(f_H(\text{Nov-15-95:0:00AM}, S' = abc, apartment_H)) = (\text{S_25, 540.00, 81.00}).$$

We can also define the navigation from one object across a relation to a second object. For example, let the object types $OT1$ and $OT2$ be related by the relationship RT. Now we select the object of $OT2$ that was related to the object of $OT1$ with surrogate s at the historical time t_H:

$$\begin{aligned} ot2_H = \pi_{A_1,\ldots,A_l}(f_H(\ & t_H, \\ & TRUE, \\ & OT2_H \underset{R_2=S'}{*} \delta_{T_H \leftarrow T'_H}(f_H(t_H, R_1 = s, RT_H)))) \end{aligned} \tag{11}$$

The inner function $f_H(t_H, R_1 = s, RT_H)$ retrieves a table containing the historical relation that linked the object s with an object of $OT2_H$ at the time t_H. The column T_H of this table is renamed to T'_H because the table of $OT2_H$ also contains a column called T_H. Then the join between the resulting table and $OT2_H$ retrieves all objects which fulfil the condition $R_2 = S'$. Finally, the second function retrieves a table containing the historical object which was related to the object s at the time t_H.

3.5 Future of Objects

Future actions manipulating objects are: insert, update, and delete. Future actions manipulating relations are: insert and delete. In the following we describe how these actions and therefore the future objects and relations are stored in the future database, and how they are applied to the application database.

For each action the transaction and the valid timestamp, a sequence number and the manipulation type is stored in the action table. Depending on the action, different data are stored in the future tables for object types and relationships:

- For an **insert** action the new object or the new relation is stored in the corresponding table of the future database. Because of that the object type tables always contain columns for all attributes (unlike the history tables which contain only attributes which are subject to historical recording).
- For a **delete** action only the surrogate of an object or — in case of a relation — the surrogates of two related objects must be stored in the future tables.
- For an **update** action the new values of all attributes concerned are stored. In this case we have the problem to distinguish between attributes that are not concerned and attributes that should be set to a null-value. Therefore, we store the information about the attributes concerned in a special column in the action table (not described further in this paper).

Periodically a special $\mathcal{L}eu$-process (see Section 4.2) is started. It executes all actions whose validity time is reached. One action is, for example, the insert

of a new object. In this case, the tuple representing the object with the future surrogate s_F can be selected from the future table OT_F of its object type. The insert action itself with its valid time v_F can be found in the action table AT_F. Now the object ot_A to be inserted into the application database can be determined as follows:

$$ot_A = \pi_{S',A_1,\ldots,A_n}(\sigma_{S_F=s_F}(OT_F) \underset{B=S_{AT}}{*} \sigma_{V_F \leq v_F \wedge M='insert'}(AT_F)) \bullet \omega \bullet \ldots \bullet \omega \quad (12)$$

Because the object does not have any relation yet, all foreign key columns are filled with null-values ω. Potential queries concerning relations do not succeed to deliver results other than empty sets. When an actions fails, it is restarted with the next execution of the $\mathcal{L}eu$-process (see Section 4.2 for implementation details).

3.6 History and Future of Objects

The concepts for dealing with the history and with the future of an object are orthogonal, i.e. both can be applied to one object without affecting each other.

Nontheless, there are two situations which have to be considered in more detail for objects whose history and whose future is managed. We assume that object *ot* is subject to historical and future recording in the following:

- For a given time t_0, a future action on *ot* is defined to be carried out at time $t_1, t_1 < t_0$ (stored in column V_F in the corresponding table). Thus, from a semantic point of view, we would not call this a future action, because its execution time is in the past. A potential reason for such a retrospective future action could be that an update of *ot* has been forgotten and that the update should be backdated. We do not support retrospective future actions due to the following reasons: Firstly, the history of an object could be arbitrarily manipulated. The *real* history can not be recognized, once retrospective actions are carried out. Secondly, even if a particular retrospective update helped to represent the real history of an object appropriately, we have to be aware of decisions and other actions performed based on the value of the object which was considered valid. That means that between time t_1 and t_0, actions may have been performed based on the version of *ot* which was valid then. When a retrospective actions manipulates *ot* at t_0 and backdates this modification to time t_1, then all actions which are based on object *ot* and which took place between t_1 and t_0 become meaningless.
- Let t_0 be the current time. We assume that a future action a_1 on *ot* is to be performed at time $t_1, t_1 > t_0$. When at time $t_2, t_0 < t_2 < t_1$ it is decided to delete the future action (i.e. not to carry it out at t_1), then a_1 is removed from the action table. The modification of the action table itself is not

recorded, because it only concerns a potential future version of *ot*, but not any actual version of *ot*. In brief, only modifications of current objects are recorded, the history of an object's future is not recorded. This is due to the fact that the future of an object is a *potential* future. We believe that it is sufficient to consider the potential future of an object at any time (in contrast to considering the history of the potential future of an object).

Based on the decision to forbid retrospective future actions and not to record the history of an object's future, we decouple the history database HDB and the future database FDB completely. Any object updates raised by future actions affect the current database ADB (but not at all the HDB), any modification of a current object concerns only the HDB.

4 IMPLEMENTATION OF FDB AND HDB

In this section we present a restricted selection of access functions useful to operate on objects and relations stored in FDB (Section 4.1) and HDB (Section 4.5). These functions are available as an *Application Programming Interface*. All these functions are based on conventional SQL. Temporal extensions as implemented in TSQL are not used. In Section 4.2 we describe the process of transforming activities defined in FDB into a valid state in ADB. This process, called *f-batch*, demands special attention regarding sequence of execution, transaction behaviour, consistency, error handling and performance. Finally, we take a look at the mechanisms responsible for creating and maintaining a complete and compact history of former versions of objects and relations (Section 4.3 and 4.4).

4.1 Access Functions to FDB

The API supports insert-, update- and delete-operations of future activities, so called *f-activities*. These f-activities define operations that are to be executed in future times on the ADB. Furthermore, there is a need for operations that permit the definition of future relations between objects and the deletion of such relations.

When working with future relations we have to consider that both source objects as well as target objects can be located in ADB or FDB. For example, it may be known that the *bank account* of a *tenant* will change in the future. If the new *bank account* already exists in ADB, then only the relation is subject to future recording. Otherwise, the new *bank account* will also be created in future times. Figure 6 summarizes the possible combinations in which objects from ADB and FDB can be interlinked by current and future relations.

Given a non-temporal database, navigation is understood as finding a re-

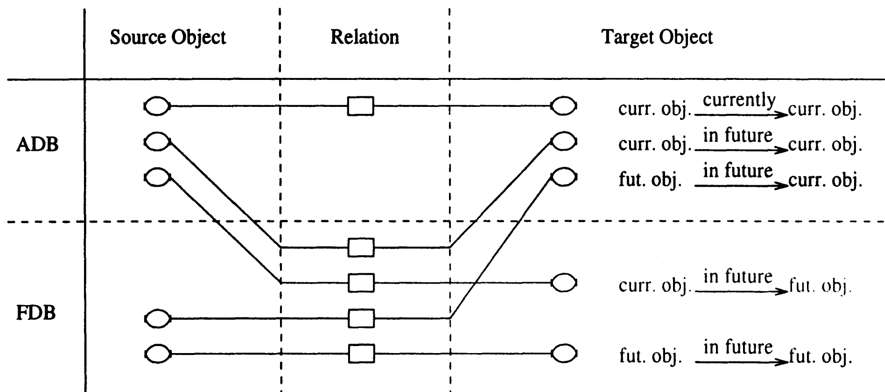

Figure 6 Relationship types

lated target object to a given source object, while both source and target are existing at present time. When dealing with future relations, the additional paths presented in Figure 6 suggest a two-step-approach to navigation. The first step is to determine all future relations to a given object — these are found in FDB. Then in a second step the related target objects are looked for. These can be members of ADB or FDB.

As already mentioned in Section 3 FDB stores future activities instead of future objects. We do not know how the ADB will change until the f-activity is performed. In consequence, no automatic consistency-checking can be done on FDB objects. Therefore, the API includes special functions to check whether a future action is executed at a given time.

4.2 Execution Hierarchy of FDB Activities

When working on ADB, activities are grouped by database transactions into atomic units. The transaction times of all objects and relations modified within one transaction are identical. This behaviour is useful for handling abstract objects, composed of several objects and relations. In order to supply a similar functionality when working with future objects and relations, we introduce so called *f-transactions*. At execution time f-transactions are translated into database transactions. For example, if a f-activity fails, all previously executed f-activities of this f-transaction are rolled back and the f-transaction is terminated with an error. If on the other hand the f-transaction can be completed without errors the newly generated ADB-state is commited and the f-transaction is marked as successful.

The user has to determine the execution sequence of f-transactions as well as the sequence of f-activities within each f-transaction. He also has to decide upon the time an f-transaction has to be carried out. For this purpose the *action table* AT_F contains the three attributes V_F, TR and SN (see Section

3.3). V_F gives the validity time at which a f-transaction is to be started. TR is a number expressing the execution sequence of all f-transactions holding the same validity time. Furthermore, all f-activities having the same TR- and V_F-attributes build a f-transaction. SN is a number which gives the execution order of f-activities of the same f-transaction. Figure 7 shows a graphical representation of the execution hierarchy.

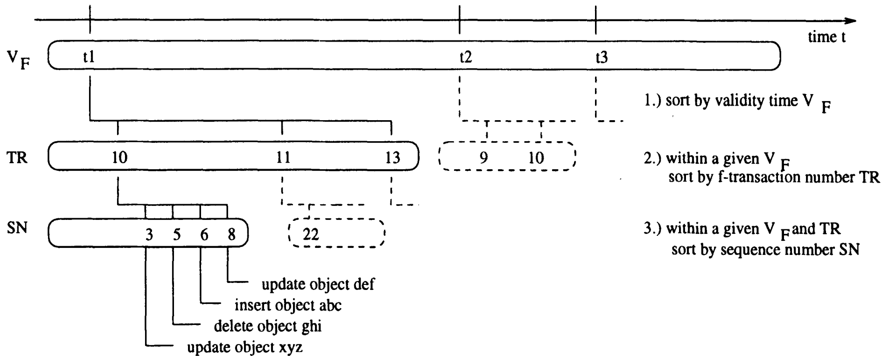

Figure 7 FDB execution hierarchy

At the top of the diagram we have three different validity times t_1, t_2 and t_3. Attribute V_F is the highest ordering criterion. At time t_1 three f-transactions numbered 10, 11 and 13 are to be executed in ascending order. F-transaction number 10 consists of four f-activities numbered 3, 5, 6 and 8 (attribute SN). Though this structure of the *action table* implies some data redundancy on the attributes V_F and TR (compare Figure 5 with $TR = 1$), the processing speed is substantially improved. By only investigating the *action table* and without consulting any OT_F- or RT_F-table we can find out which f-transactions and f-activities are to be executed and in which order. The corresponding SQL-statement is formulated as follows:

SELECT	$*$	FROM	AT_F
WHERE	$V_F <=$ current time	ORDER BY	V_F, TR, SN;

As soon as the validity time V_F of a f-transaction is reached its state is changed from *waiting for execution* (WFE) to *ready for execution* (RFE). These states are not registered as real attributes in any table. They are implicitly derived from the validity time V_F. In periodical intervals the f-batch checks the state of f-transactions, changes them from WFE to RFE if necessary and executes all f-transactions in RFE-state. An upper limit for the potential duration of the batch and its execution frequency can be configured. This limit is necessary to avoid collisions with other database activities like computer or database shutdowns. If the upper limit for the f-batch duration avoids that certain f-transactions (being in RFE-state) are actually executed,

then these f-transactions remain in RFE-state, until the f-batch is started for the next time. The unit of time, in which the validity time V_F is expressed, is derived from the period between two f-batch executions. For example, if the f-batch is only activated every 24 hours, the resolution of V_F is one day. Hours, minutes and seconds specified in V_F are not considered.

4.3 Mechanism for Creating Histories

Every operation on objects or relations in ADB causes the new object or relation to be copied immediately to HDB, provided it is subject to historical recording. This job is done by database triggers as available in the RDBMS Oracle. Depending on the supported database system one or more triggers are implemented on every OT_A- and RT_A-table. Each trigger implicitly has knowledge of the type of manipulation it is responsible for. So, when being fired, it can set the M-attribute of the OT_H- and RT_H-tables directly. As the transaction time T_H is unknown at this moment — T_H is determined with the COMMIT-command — it is set to ω (null-value).

$\mathcal{L}eu$ itself is responsible for the correct setting of transaction time T_H. It records every ADB table being touched by the current transaction. When a COMMIT is requested, all corresponding HDB-tables are updated by replacing ω-values in the T_H-attribute with the actual commit time. Thereafter the transaction is commited. In case the transaction is rolled back the database itself removes all obsolete entries in HDB created by any trigger. No further processing has to be done.

With the mechanisms described up to now a negative side-effect is encountered. The history tables are populated not only by commited states but also by transient states which exist only temporarily in the course of executing transactions. To avoid this, the trigger procedure tests whether the current transaction has already created an entry in the affected history table. This can be identified by $T_H = \omega$. If so, the present entry (no more than one can exist) is replaced by or merged with the new one. Section 4.4 describes in detail in which cases and how the process of merging is done.

4.4 Lifespans in HDB

The start and end of objects' and relations' lifespans is captured in the attributes M and T_H of the OT_H- and RT_H-tables. The manipulation type M has to be interpreted appropriately as shown in Figure 8.

The attributes *Truncate* and *R_Truncate* indicate that the history of an object (*Truncate*) or relation (*R_Truncate*) has either been removed from the database or has never been available to the database. An example for their usage is given in 4.5.

Manipulation M	Meaning
Insert	Start of existence of object and of all its attributes
Delete	End of existence of object and of all its attributes and of all relations kept in this OT_H-table
Update	End of old attribute states and begin of new attribute states
Truncate	First known appearence of object in ADB, but no knowledge about real start of existence (e. g. imported data)
R_Insert	Start of existence of relation and eventually of object
R_Delete	End of existence of relation
R_Truncate	First known appearence of relation in ADB

Figure 8 Manipulation types M

A *R_Insert*-entry can mark the creation of a new relation and at the same time the creation of a new object. For 1:1- and 1:n-relationships the relation is kept in the OT_H-table (see e. g. Figure 4 column R_lease). If within the same transaction both a new object and its relation are created then the *Insert*-entry is removed from the history table. The remaining *R_Insert*-entry holds all the information required to represent the new situation correctly. In consequence the number of entries per transaction can be limited to one.

The same algorithm is applied when deleting relations (*R_Delete*) and related objects (*Delete*) within one transaction. In this case, the *Delete*-entry signals both the end of the object's and the relation's lifespan whilst the *R_Delete*-entry is removed from the history table.

Another exception is an object being created (*Insert*) and updated (*Update*) within one transaction. The *Update*-entry represents the correct object state and therefore is preserved. To document that the object has been newly created, the *M*-attribute of the *Update*-entry is changed to *Insert* and the former *Insert*-entry is removed.

Finally, we have to deal with situations where an object is created (*Insert*) and deleted (*Delete*) within the same transaction. In ADB such an object is never visible to other transactions. Therefore, the creation of a historical object has to be prevented, too. When an object is deleted, the trigger checks if the corresponding *Insert*-operation on this object took place within the current transaction ($T_H = \omega$) or before. In the former case both the *Insert*- and *Delete*-entries are removed from the histroy table. Figure 9 summarizes these cases.

The first column lists the two activities executed within a single transaction on the same object. The second column shows which entries — described by the attributes M, R_1 and A_1 — are temporarily created in the history table and the last column depicts which entry is actually fixed in the history table.

In all cases the database triggers and the implemented algorithms are the

Operation in ADB	Intra-transaction HDB-situation			Merged (committed) HDB-situation		
	M	R_1	A_1	M	R_1	A_1
1. Create Object 2. Create Relation	Insert R_Insert	ω xxx	<*any*> <*any*>	R_Insert	xxx	<*any*>
1. Delete Relation 2. Delete Object	R_Delete Delete	ω ω	<*any*> <*any*>	Delete	ω	<*any*>
1. Create Object 2. Update Object	Insert Update	<*any*> <*any*>	aaa bbb	Insert	<*any*>	bbb
1. Create Object 2. Delete Object	Insert Delete	<*any*> <*any*>	<*any*> <*any*>			

Figure 9 Merging of intra-transaction-states

only ones manipulating historical objects. The user can only select them. Furthermore, retroactive updates are not supported, because the system is a commercial information system in which all manipulations and their actual timestamps must be monitored. It is not allowed to undo or change any modifications of objects.

4.5 Access Functions to HDB

To evaluate the information stored in HDB, special access functions are made available that allow snapshot views as well as timespan views. In addition to this retrieval functionality, a delete mechanism exists to systematically remove the complete history of an object or parts of it. This is useful for optimizing database space and improving processing speed.

Given a time limit t_{limit}, the delete machanism removes the history of all selected objects and relations whose transaction time T_H is older than or equal to t_{limit}. Thereafter, for every modified object/relation a new entry is inserted into the concerned OT_H- and RT_H-tables. This entry indicates the removal of part of the history by setting the M-attribute to *Truncate* or *R_Truncate* and the transaction time T_H to t_{limit}.

A similar situation is encountered when an object type which is already containing objects becomes subject to historical recording. There is no information about the past of the objects, e. g. their creation times. Therefore, an initial entry is generated in the history table for every object. Its transaction time T_H is set to the current time and its attribute M is set to *Truncate*.

5 EXPERIENCES AND CONCLUSION

Our experiences in developing commercial information systems have raised the idea to consider temporal aspects of objects on the conceptual level. Without doing so, individually managing temporal aspects for particular object types costs a high price in terms of development effort. In addition, it usually leads to heterogeneous solutions which impact maintainability.

Once we integrated modeling of temporal aspects into entity–relationship modeling, a substantial portion of object types was modeled as being subject to historical recording and/or future modification recording. This extensive use of the modeling constructs resulted in an increased data volume and in a decreased database performance. This experience is based on the development of large information system for the housing construction and administration area (Gruhn and Wolf 1994). We believe that the use of temporal aspects in entity-relationship modeling could be beneficial in developing other information systems as well.

The remedy was a guideline which helped to identify which object types and which attributes of which object types actually have to be considered in modeling temporal aspects. Even though this guideline is rather simple, it helped to enforce conscious decisions about temporal aspects. In the end, less than 10% of all object types are subject to historical recording and/or future modification recording.

REFERENCES

Ait-Braham, A., Theodoulidis, B. and Karvelis, G.: 1994, *Conceptual Modelling and Manipulation of Temporal Databases*, *in* P. Loucopoulos (ed.), *Proceedings of the* 13th *International Conference on the Entity-Relationship Approach*, Springer, Manchester, UK, pp. 296–313. Appeared as Lecture Notes in Computer Science no. 881.

Bhargava, G. and Gadia, S.: 1993, *Relational Database Systems with zero information-loss*, *IEEE Transactions on Knowledge and Data Engineering* **5**(7), 76–87.

Clifford, J. and Crocker, A.: 1993, *The Historical Relational Data Model (HRDM) Revisited*, *in* A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass (eds), *Temporal Databases*, The Benjamin/Cummings Publishing Company, Redwood City, CA, US, pp. 6–26.

Clifford, J., Croker, A. and Tuzhilin, A.: 1994, *On completeness of historical relational query languages*, *IEEE Transactions on Database Systems* **19**(1), 64–116.

Dinkhoff, G., Gruhn, V., Saalmann, A. and Zielonka, M.: 1994, *Business Process Modeling in the Workflow Management Environment LEU*, *in* P. Loucopoulos (ed.), *Proceedings of the* 13th *International Confer-*

ence on the Entity-Relationship Approach, Springer, Manchester, UK, pp. 46–63. Appeared as Lecture Notes in Computer Science no. 881.

Gadia, S.: 1988, *A homogenous relational model and query languages for temporal databases, IEEE Transactions on Database Systems* **13**(4), 418–448.

Gadia, S. and Vaishnav, J.: 1985, *A query language for a homogenous temporal database, Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 51–56.

G.Kappel, Pröll, B., Rausch-Schott, S. and Retschitzegger, W.: 1995, *TriGSflow Active Object-Oriented Workflow Management, Hawaii International Conference on System Sciences (HICSS'95)*, Hawaii, USA.

Gruhn, V., Pahl, C. and Wever, M.: 1995, *Data Model Evolution as a Basis of Business Process Management, OOER95: Object-Oriented and Entity-Relationship Modeling*, Springer, Berlin, Gold Coast, Australia, pp. 270–281. Appeared as Lecture Notes in Computer Science 1021.

Gruhn, V. and Wolf, S.: 1994, *Software-Entwicklung auf der Basis von Geschäftsprozeß-Management, Theorie und Praxis der Wirtschaftsinformatik* **31**(180), 116–125.

Gruhn, V. and Wolf, S.: 1995, *Software Process Improvement by Business Process Orientation, Software Process Improvement and Practice* **1**, 49–56. Pilot Issue.

Hou, W. and Özsoyoglu, G.: 1993, *Processing real-time aggregate queries in CASE-DB, IEEE Transactions on Database Systems* **18**(2), 224–261.

Jensen, C., Clifford, J., Gadia, S., Segev, A. and Snodgrass, R.: 1994, *A Consensus Glossary of Temporal Database Concepts, SIGMOD Record* **23**(1), 52–64.

Jensen, C., Soo, M. and Snodgrass, R.: 1989, *Unifying Temporal Models via a Conceptual Model, Information Systems* **19**(7), 513–547.

Özsoyoglu, G. and Snodgrass, R.: 1995, *Temporal and Real-Time Databases: A Survey, IEEE Transactions on Knowledge and Data Engineering* **7**(4), 513–532.

Pissinou, N., Snodgrass, R., Elmasri, R., Mumick, I., Ozsu, M., Percini, B., Segev, A. and Theodoulidis, B.: 1994, *Towards an Infrastructure for Temporal Databases: Report on Invitational ARPA/NSF Workshop, Technical Report TR 94-01*, University of Arizona.

Snodgrass, R.: 1995, *The TSQL2 Temporal Query Language*, Kluwer Academic Publishers.

Snodgrass, R. and Ahn, I.: 1985, *A Taxonomy of Time in Databases, Proceedings of the Fourth ACM SIGMOD*, New York, NY, US, pp. 236–246.

Snodgrass, R. and Ahn, I.: 1986, *Temporal Databases, IEEE Computer* **19**, 35–42.

Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A. and Snodgrass, R.: 1993, *Temporal Databases*, The Benjamin/Cummings Publishing Company, Redwood City, CA, US.

Wieringa, R., de Jonge, W. and Spruit, P.: 1994, *Roles and dynamic subclasses: a modal logic approach*, *in* M. Tokoro and R. Pareschi (eds), *Proceedings of the European Conference on Object-Oriented Programming, Bologna*, Springer, Berlin, pp. 32–59. Appeared as Lecture Notes in Computer Science no. 821.

6 BIOGRAPHY

Martin Fegert studied Medical Informatics at the University of Heidelberg, Germany and received the degree of 'Diplom Informatiker der Medizin' in 1995. Since then he has been employed as software engineer by the company o.tel.o (formerly named LION) working in the field of relational databases applications. He participated in the design and implementation of temporal features in an existing huge database application.

Volker Gruhn received a MS degree (Diplom-Informatiker) (1987) and a Ph.D in Computer Science (1991) from the University of Dortmund. From 1991 to 1996 he has been responsible for the software engineering department of LION. From 1994 to 1996 he was chief technical officer of LION. Since 1997 he is associate professor for software technology at the University of Dortmund. Volker Gruhn has authored and co-authored about 50 articles which appeared in national and international journals and conference proceedings. His major professional interests are software processes, workflow management and database systems for software engineering environments. He is member of IEEE and ACM.

Monika Schneider received a MS degree (Diplom-Informatikerin) in Computer Science (1994) from the University of Dortmund. From 1994 to 1997 she has been employed as software engineer by o.tel.o (formerly named LION) working in the field of relational database applications. She is currently working as senior consultant in the field of database applications at sd&m.

Acknowledgements: We would like to thank all members of the $\mathcal{L}eu$ team for their cooperation in implementing these strategies. Our special thanks are due to Michael Zielonka, who is heading the database team. Moreover, we would like to thank Hans Werner Schmitz and Juri Urbainczyk for their comments on an earlier version of this article.